KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering

Modelling Diversity in Software Product Lines

Radu Muschevici

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor in Engineering

December 2013

Modelling Diversity in Software Product Lines

Radu MUSCHEVICI

Supervisory Committee:
Prof. Dr. Hugo Hens, chair
Prof. Dr. Dave Clarke, supervisor
Prof. Dr. ir. Frank Piessens, co-supervisor
Prof. Dr. ir. Marc Denecker
Prof. Dr. ir. Wouter Joosen

Prof. Dr.-Ing. Ina Schaefer
(TU Braunschweig)
Prof. Dr. ir. Eric Steegmans
(KU Leuven)

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor in Engineering

December 2013

D/2013/7515/141
ISBN 978-94-6018-756-8

Abstract

This dissertation presents the results of a line of research into modelling the diversity inherent to software product lines. We approach diversity modelling from two different angles: via a modelling formalism that allows reasoning about properties of (parts of) the modeled system, and via a programming language (complete with compiler and runtime environment) that allows the implementation and execution of entire SPLs.

Diverse systems are prevalent in the modern software landscape, as they can be readily adapted to meet variable user requirements. Diversity adds significant complexity to software, which is best managed using adequate variability modelling techniques as part of a dedicated engineering process for diverse software. Furthermore, it is important that the software's variability is modelled consistently throughout the development life cycle, thus avoiding discrepancies between models used at different stages of development or by different stakeholders.

Software product lines (SPL) are diverse systems that are developed using a dual engineering process: (a) *family engineering* defines the commonality and variability among all members of the SPL, and (b) *application engineering* derives specific products based on the common foundation combined with a variable selection of features. The number of derivable products in an SPL can be exponential in the number of features. This inherent complexity poses two main challenges when it comes to modelling: Firstly, the formalism used for modelling SPLs needs to be modular and scalable. Secondly, it should ensure that all products behave correctly by providing the ability to analyse and verify complex models efficiently. The choice of a system modelling formalism that is both expressive and well-established is therefore essential. In this thesis we propose to extend Petri nets to *Feature Petri Nets*, or *feature nets* (FN) for short. Petri nets provide a framework for formally modelling and verifying single software systems. Feature nets offer the same sort of benefits for software product lines. We show how SPLs can be modelled in an incremental, modular

fashion using feature nets, and also provide a feature nets variant that supports modelling dynamic software product lines.

The Abstract Behavioural Specification (ABS) language and tool suite constitute a platform for developing and analysing highly adaptable concurrent software systems. In line with these goals, we extend ABS to support variability modelling. We provide four language extensions that are used together to describe the variability of the system under development and ensure its consistency: μTVL is used for describing feature models at a high level of abstraction, the Delta Modelling language describes variability of the code base in terms of deltas, the Product Line Configuration language is for linking feature models and deltas together and the Product Selection language is for describing a specific product to extract from a product line.

Dynamic software product lines (DSPLs) combine the advantages of traditional SPLs, such as an explicit variability model connected to an integrated repository of reusable code artifacts, with the capability to exploit a system's variability at runtime. When a system needs to adapt, for example to changes in operational environment or functional requirements, DSPL systems are capable of adapting their behaviour dynamically, thus avoiding the need to halt, recompile and redeploy. Our contribution to the emerging field of DSPL engineering is the extension of the ABS language and execution environment to support developing and running dynamic SPLs. Systems developed using ABS are compiled to Java, and are thus executable on a wide range of platforms.

The SPL approach to software development has gained a significant level of academic interest and industrial adoption. It is likely to attract even more attention as diverse software systems become the norm rather than the exception. The work presented in this thesis contributes to the understanding of variability as an explicit concern in software development, and offers concrete solutions for managing variability by providing models and modelling tools for variability.

Beknopte samenvatting

Dit proefschrift beschrijft de resultaten van een onderzoek gefocust op het modelleren van de heterogeniteit inherent aan software productlijnen (SPLs). Onze aanpak benaderd het probleem vanuit twee verschillende aspecten. In de eerste aanpak gebruiken we een modelleerformalisme dat ons toelaat over de eigenschappen van een SPL te redeneren. In de tweede aanpak ontwikkelen we een programmeertaal die het mogelijk maakt om een SPL te implementeren en uit te voeren.

Heterogene software systemen zijn veel voorkomend in het moderne software landschap, waar ze voornamelijk gebruikt worden om tegemoet te komen aan sterk variërerende gebruikers eisen. Deze heterogeniteit maakt software systemen veel complexer. Deze complexiteit wordt best beheerd met behulp van modelleer technieken die geschikt zijn voor hoge variabiliteit. Om te vermijden dat er discrepanties mogelijk zijn tussen de modellen van de verschillende ontwikkeling fases of die van de verschillende belanghebbenden, is het belangrijk dit modelleren consequent wordt toegepast tijdens het ontwikkelingsproces.

Software product lijnen zijn heterogene systemen die ontwikkeld worden met behulp van een dubbel engineeringproces: (a) *Familie engineering* specificeert de gemeenschappelijkheid en variabiliteit tussen all leden van de SPL, en (b) *applicatie-engineering* bepaalt de specifieke producten op basis van een gemeenschappelijke fundering gecombineerd met een variabel aantal functies. De complexiteit inherent aan SPLs veroorzaakt twee uitdagingen op het gebied van modelleren: ten eerste, het formalisme dat gebruikt word voor het modelleren moet modulair en schaalbaar zijn. Ten tweede, datzelfde formalisme moet er ook voor zorgen dat all producten zich correct gedragen met behulp van een optie om complexe modellen te analyseren en te verifiëren. De keuze voor een modelleerformalisme dat zowel expressief als goed gevestigd is, is dus essentieel. In deze thesis stellen we voor om Petri nets uit te breiden tot *Feature Petri Netten*, of *feature nets* (FN) in het kort. Petri nets bieden een framework aan waarbinnen een enkelvoudig software systeem gemodelleerd en geverifieerd kan

worden. Feature nets bieden dezelfde soort functionaliteit aan voor software product lijnen. In deze thesis tonen we aan hoe SPLs gemodelleerd kunnen worden op een incrementele en modulaire wijze met behulp vanfeature nets, en bieden we ook een variant op feature nets aan die kan gebruikt worden voor het modelleren van dynamische software product lines.

The Abstract Behavioural Specification (ABS) taal en tool chain vormen een platform voor het ontwikkelen en analyseren van erg aanpasbare meerdradige software systemen. In lijn met deze doelstellingen, hebben we ABS uitgebreid met de mogelijkheid om variabiliteit te modelleren. We presenteren vier programmeertaal uitbreidingen die, wanneer ze te samen gebruikt worden, de variabiliteit van het systeem onder ontwikkeling beschrijven en de consistentie van het systeem waarborgen: μTVL wordt gebruikt om feature modellen op een abstracte wijze te beschrijven, de Delta Modelling taal beschrijft de variabiliteit van de broncode in termen van delta's, de Product Line Configuration taal wordt gebruikt om feature modellen en delta's te verbinden en de Product Selection taal beschrijft het specifieke product dat uit de product line wordt afgeleid.

Dynamische software product lijnen (DSPLs) combineren de voordelen van traditionele SPLs, zoals het verbinden van expliciete variabiliteit modellen met geïntegreerde databases van herbruikbare code artefacten, met de mogelijkheid om de variabiliteit van een systeem aan te passen tijden de uitvoering ervan. Wanneer een systeem zich moet aanpassen, om bijvoorbeeld te beantwoorden aan veranderingen in zijn omgeving, dan zijn DSPL systemen in staat om hun gedrag dynamisch aan te passen, waarmee ze zo vermijden dat het systeem wordt stilgezet, gecompileerd en herstart. Onze contributie aan het opkomende veld van DSPL engineering is de eerder beschreven uitbreiding aan de ABS taal en een uitvoering omgeving voor het ontwikkelen en uitvoeren van DSPLs. Software systemen die ontwikkeld zijn met behulp van ABS worden gecompileerd naar Java en zijn dus uitvoerbaar over een groot aantal platformen.

Het gebruik van SPLs bij het ontwikkelen van software systemen kan reeds rekenen op een aanzienlijke mate van academische belangstelling en industriële adoptie. Naarmate dat software systemen steeds heterogener worden zal deze belangstelling steeds groter worden. Deze thesis draagt bij tot het beter verstaan van variabiliteit als een belangrijke zorg bij het ontwikkelen van software systemen, en biedt concrete oplossingen aan voor het beheersen van deze variabiliteit met behulp van modellen en modelleer tools voor variabiliteit.

Acknowledgements

First and foremost, I would like to express my gratitude towards my advisor Dave Clarke. Dave, thanks for offering me the opportunity to do a PhD at KU Leuven, for your advice and your constant support and encouragement during four and a half years. I wish many more students will have the luck to do science with you.

Many thanks José Proença for your day-to-day guidance, for sharing your knowledge with me and for always taking the time to answer my questions. Also thanks for all your help during my restless stay in Leuven.

I am grateful to the members of my jury: Marc Denecker, Wouter Joosen, Frank Piessens, Ina Schaefer and Eric Steegmans for attentively reading the text and providing useful comments and feedback. Thanks to Prof. Hugo Hens for chairing my preliminary defence and to Prof. Carlo Vandecasteele for chairing my public defence.

Being part of the HATS project, I had the great pleasure and privilege to work together with several brilliant researchers. These collaborations have certainly been the defining aspect of my doctorate. I thoroughly enjoyed meeting and doing research with you, and I learned a tremendous lot in this process. Thank you Reiner Hähnle, Einar Broch Johnsen, Rudi Schlatte, Peter Y.H. Wong, Yannick Welsch, Jan Schäfer, Michiel Helvensteijn, Richard Bubel, Volker Stolz, Elvira Albert and Michäel Lienhardt.

Thanks to my friends and colleagues in DistriNet, especially to Dimiter, Rula, Marco, Mario, Adriaan and Ilya for sharing the fun of doing a PhD.

With the CS rock climbing group I enjoyed many fun and relaxing after-work climbing sessions, which have probably played an important part in keeping me sane. Thanks to the regulars for making sure that hardly any Wednesday went by without the opportunity to exercise: Mika, José, Wouter, Jeroen, Stefan, Bart and Nelis.

I would like to extend my thanks to all people in the CS department that work tirelessly to ensure that all administrative issues are taken care of, especially to Marleen Somers and Esther Renson, Katrien Janssens and Ghita Saevels, Anita Ceulemans and Bart Swennen.

Last but not least, I would like to thank my family: my parents Dan and Liana, and my sister Cristina for being there when I need them, providing the kind of tacit spiritual backing that is rarely acknowledged, yet is essential for moving forward. Thanks to Mika for a true friendship and for being there for me in difficult and in happy moments. Thanks to Rachel for keeping my friendship through time and space, for ever encouraging me and helping me overcome my doubts.

The research presented in this thesis was, for the most part, funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods (`http://www.hats-project.eu`). I am grateful for the provided financial support.

Contents

List of Figures

Chapter 1

Introduction

Diversity is a widespread characteristic of contemporary software. A diverse software system offers potential customers or users a wide range of system variants to choose from. The variants offered differ in terms of the features they provide or the application context they support, and are thus able to satisfy the requirements of a larger number of customers than a single system with a fixed set of features.

Diverse systems are more complex and thus more challenging to develop than single systems because of the multitude of variants that they include. Hence the capabilities of these variants and the relationships among them need to be managed using explicit *models*.

1.1 Software Variability Modelling

Software diversity is the result of an attuned software development process that takes into account variable user requirements and ensures that they are addressed consistently during design, implementation and validation phases. This means that variable requirements entail variable analysis, design, implementation and validation artifacts. Hence a software development methodology for diverse systems needs adequate means to represent and organise variable development artifacts. The process of representing and organising variable software development artifacts is known as *variability modelling*.

A variability model defines relationships among variable development artifacts in order to describe all variants of a software system. Variability models can be

classified as belonging to either the *problem space* or to the *solution space* [103]. Problem space variability models are formulated around the needs of customers and users, using the vocabulary and paradigms familiar to these stakeholders, and focus on the requirements expressed in a software development project. An established [16] way of modelling problem space variability, which we also adopt in this line of work, is feature-oriented variability modelling, also know as feature modelling [31, 13].

Solution space variability models are defined in terms of development artifacts such as architectural or implementation-related components. In this thesis, problem space variability is modelled following the delta-oriented paradigm [101], a recent extension of feature-oriented programming [95, 11] that relies on *deltas* to describe modification to a core program.

To ensure that the implemented solution matches the user's problem, problem and solution space variability models have to be kept mutually consistent. We call the connection that keeps variability models in sync *configuration*. The configuration also ensures traceability along the development chain, which is a prerequisite for automation: given a set of requirements, the appropriate program variant can be generated automatically from the set of implementation artifacts.

Modelling variability is a central concern when developing diverse software systems. Therefore, any development methodology targeted at engineering diverse systems needs to provide facilities to manage variability in each development phase.

1.2 Software Product Lines

Software product line (SPL) engineering is a methodology for developing diverse systems. It achieves this potential by addressing variability explicitly throughout all development phases, from requirements elicitation to deployment and in some cases even at runtime, when the system is in operation. A software product line or software product family is backed by a variability model that describes how the product variants relate to each other by defining the commonality and variability among all them. The essence of the SPL engineering approach is to decompose a set of related software products into a set of finer grained artifacts (e.g. features, concepts, concerns, code fragments) and to provide a variability model, that is, a precise model of the dependencies between these artifacts. By combining artifacts together in ways that satisfy the variability model, a diverse range of system variants can be obtained. A key aspect of this approach is that it promotes and facilitates the reuse of artifacts across variants, meaning that

SPLs can be developed at lower cost, in shorter time and with higher quality than developing each variant individually [94].

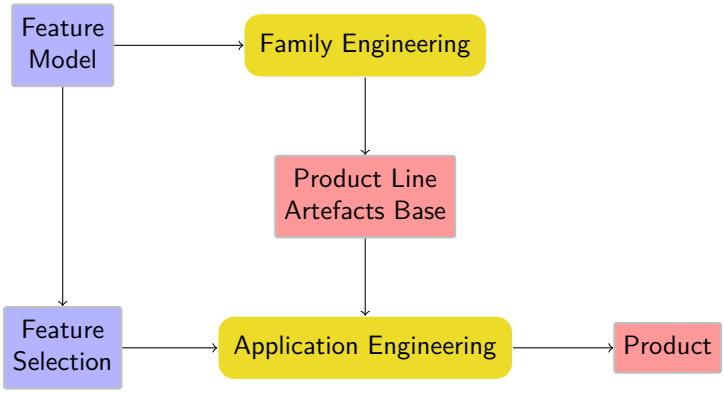


Figure 1.1: The software product line engineering method

The approach outlined above requires a two-phase software engineering process: first, the decomposition of program variants into fine grained development artifacts is known as *family engineering*. Family engineering defines the commonality and variability among all members of the SPL. Second, the process of combining development artifacts in order to derive specific products is called *application engineering*. Figure 1.1 illustrates the two engineering processes in SPL development.

In this dissertation we develop a set of methodologies and technologies that enable the modelling of diverse systems as software product lines. We start by introducing a Petri net-based formalism aimed at enabling formal reasoning about diverse systems. We then shift our focus to programming languages and extend the Abstract Behavioural Specification (ABS) language with dedicated support for variability modelling. In the final step we enhance our SPL modelling language and tools to represent runtime variability, thus enabling the development of executable, dynamic software product lines.

1.3 The ABS Language

ABS, the Abstract Behavioural Specification language [122], is a recent specification and programming language that positions itself between between design-oriented and implementation-oriented specification languages. It allows the precise modelling and analysis of concurrent systems, focusing on their

functionality while abstracting from concerns such as concrete resources, deployment scenarios and scheduling policies [69]. ABS is the core outcome of the European project HATS (Highly Adaptable and Trustworthy Software using Formal Methods).

On the surface ABS resembles standard object-oriented programming languages such as Java, both by using imperative constructs to model control flow and by supporting class-based object-oriented programming. However, class inheritance is not supported, as code reuse is achieved using the SPL engineering approach (cf. Section 1.2). ABS also supports functions and algebraic data types to enhance its abstract modelling capabilities. The most important difference between ABS and Java is in the underlying concurrency model: ABS is based on active object-concurrency [3]. ABS models are compiled to Java and execute on the JVM. A complete description of all ABS features can be found in the ABS language specification [1].

The SPL modelling framework developed in Chapters 3 and 4 uses the ABS language and tool suite as a starting point and extends it to support SPL and DSPL development. ABS is a favourable platform for implementing these goals because it offers standard high-level constructs such as algebraic data types, functions and objects, to make writing *behavioural models* easier. Moreover, these models are *executable* by way of compilation to standard Java.

1.4 Problem Statement

Diversity is an important concern in all areas of industrial production. Beginning in the 1970s, studies aimed at identifying factors that increase consumer satisfaction [83, 96] have revealed that consumers have highly diverse preferences and needs. Consequently, their varied preferences need to be addressed with products to match. In line with these findings came the realisation that searching for the *universal* product, i.e., the one product that satisfies most customers is misguided. Instead, from a producer or marketer's point of view, the key to higher consumer happiness is to have a range of "perfect products" on offer.

Nowadays the application of these findings pervades our everyday lives as consumers: whenever we go shopping, no matter it being for something as mundane as a spaghetti sauce for today's dinner, or for a longer-term acquisition such as a car, we have to first decide upon a particular product variant out of a product line of seemingly very similar products. While the steady increase in the number of variants, and thus choices, has recently been associated with a backlash in consumer satisfaction [108], there is no doubt that variable needs

and requirements are best addressed with a varied range of solutions, each closely adapted to address the respective problem.

In software, the need to manage diversity across a set of similar programs has been recognised as early as the late 1970s [92], leading up to the formation of today's expansive software product lines research field.

Software permeates virtually all aspects of life. Formal modelling and verification methods help creating software we can trust to function as specified. This is especially true for SPLs: The dual engineering process, which has the developer create a set of reusable artifacts and a model of the relationships between these artifacts, increases the software engineer's conceptual distance to the end product. Moreover, the combinatorial explosion of possible products leads to a huge number of different behaviours, which cannot be covered efficiently with traditional validation methods. Modelling and analysing the behaviour of entire software product lines with the help of formal methods is therefore a widely acknowledged concern in the SPL research field. To address these challenges, a central goal of the research presented in this thesis has been to develop a formal modelling and analysis framework for SPLs.

To effectively support the development of SPLs, languages and tools specific to the SPL engineering domain are needed. As SPL engineering explicitly deals with requirements that are mapped to features and then to reusable design and implementation artifacts, a comprehensive SPL development environment will necessarily represent these concepts using first-class elements of the development process. Moreover, traceability of these elements along development phases is necessary.The second central goal of our research is therefore the specification and implementation of programming language constructs that comprehensively and effectively support the development of SPLs.

While the configuration of a diverse software system was initially a development time concern, it is now increasingly associated with the runtime. There are undeniable advantages to be able to reconfigure a software system while it is running, without interrupting its service. Dynamic software product lines are expected to deliver these advantages, yet the theoretical and practical challenges raised by dynamic reconfiguration cannot be currently considered solved. Such challenges are related for example to ensuring that a program reaches a quiescent state before reconfiguration, or to automatically transferring the execution state when the program's structure is updated. Additionally, from a software engineering perspective, development tools need to provide adequate abstractions that help the programmer model and reason about dynamic updates. Our last objective is therefore to explore and address some of the unsolved challenges associated with dynamic reconfiguration, specifically by proposing a language and runtime system for dynamically adaptable software.

1.5 Outline and Contributions

This thesis is split into three main chapters. We outline the motivation behind each chapter's subject-matter and highlight contributions.

Chapter 2 presents a line of research into using Petri nets to model the behaviour of software product lines. The outcome of this research is a modelling formalism called *feature nets* that enables the specification of the behaviour of an entire software product line (a set of systems) in one single model. The behaviour of a feature net is conditional on the features appearing in the product line.

The feature nets formalism addresses the fact that many variability-intensive systems are safety-critical and thus their behaviour needs to be modelled with rigour and is subject to verification. Since our formalism is based on Petri nets, a wealth of established analysis and verification techniques [84] are at our disposal. Feature nets are also modular, meaning that a typically large SPL can be modelled incrementally, e.g. one feature at a time. By following certain modelling criteria, the behaviour of the small individual nets is guaranteed to be preserved when these are combined together to a model of the entire SPL.

Chapter 3 shifts the focus of our SPL research towards programming languages. A programming language is required in order to make behavioural models executable, and thus applicable in practice. Few integrated development environments for SPLs exist so far and those that exist do not allow the modelling of variability as a central concern, traceable throughout development phases.

We extend the ABS language to provide language constructs for encoding a feature model and a delta model in ABS, and we connect these two models of problem and solution space variability through a configuration language. Having thus established traceability between the two spaces, we develop a suite of tools that are able to analyse SPL models (i.e. feature model analysis) and automatically generate executable product variants based on a selection of features.

Chapter 4 builds on the ABS framework for static modelling of software product lines developed in the previous chapter and extends it to support models of dynamic SPLs (DSPLs). With traditional SPL engineering methods, variability is a concern tied strictly to the development phase. Deployed software

products thus lack the ability to adapt at runtime in response to changes in their execution environment or to user requirements.

The work presented in this chapter addresses the problem of runtime reconfiguration. Our contributions include the extension of the ABS variability language towards managing variability at runtime; a metaprogramming interface that enables the auto-reconfiguration of running ABS systems; an adaptive runtime environment in which DSPLs are executed; and an ABS compiler back-end that generates adaptive Java code.

Chapter 5 concludes this dissertation and outlines several opportunities for continuing this line of research.

Chapter 2

Feature Nets

2.1 Introduction

The need to tailor software applications to varying requirements, such as specific hardware, markets or customer demands, is growing. If each application variant is maintained individually, the overhead of managing all the variants quickly becomes infeasible [94]. Software Produce Line Engineering (SPLE) is seen as a solution to this problem. A Software Product Line (SPL) is a set of software products that share a number of core properties but also differ in certain, well-defined aspects. The products of an SPL are defined and implemented in terms of *features*, which are subsequently combined to obtain the final software products. The key advantage hereby over traditional approaches is that all products can be developed and maintained together. A challenge for SPLE is to ensure that all products meet their specifications without having to check each product individually, by checking the product line itself. This raises the need for novel SPL-specific formalisms to model SPLs and reason about their properties.

This chapter presents a line of research into using Petri nets to model the behaviour of software product lines. Petri nets [84] provide a solid formal basis for system modelling. They have been studied and applied widely, and they come with a wealth of formal analysis and verification techniques. The modelling formalism we develop is *feature Petri nets*, or *feature nets* (FN) for short. Feature nets are a Petri net extension that enables the specification of the behaviour of an entire software product line (a set of systems) in one single model. The behaviour of a FN is conditional on the features appearing in the

product line. The ability to model the behaviour of a set of systems in a single model brings us closer to the goal of reasoning about multiple systems in a practical way.

We extend Petri nets in three steps. We start by guiding the execution of a Petri net based on the feature selection. We call this model *transition-labelled feature nets* (FN) because it conditions the firing of transitions on the feature selection. In the second step we introduce a mechanism to dynamically update the feature selection based on the execution of the Petri net. This model is called *dynamic feature nets* (DFN). In a third step we improve upon the original FN definition with the aim of supporting net composition. The improved model allows us to develop a technique for constructing larger feature nets from smaller ones to model the addition of new features to an SPL. The feature selection is now associated with Petri net *arcs*, hence this model is called *arc-labelled feature nets*. We provide correctness criteria for ensuring that the composition of arc-labelled FNs preserves the behaviour of the original model(s).

Our three feature net models provide an elegant separation between behaviour, modelled by the underlying Petri net, and available functionality, modelled by feature sets. This work was carried out in collaboration with José Proença and Dave Clarke and was published as *Feature Petri Nets* at the 1st International Workshop on Formal Methods in Software Product Line Engineering (FMSPLE 2010) [85] and as *Modular Modelling of Software Product Lines with Feature Nets* at the 9th International Conference on Software Engineering and Formal Methods (SEFM 2011) [87, 88].

The chapter is structured as follows. Section 2.2 illustrates the modelling challenge in SPL engineering with an example, thereby motivating the need for feature nets. Section 2.3 provides the necessary background on Petri nets. Section 2.4 presents *transition-labelled* feature nets, our original formalism [85] for modelling SPL, along with their dynamic variant for modelling DSPL. In Section 2.5 we improve the original FN definition with the goal to better support modular composition, presenting *arc-labelled* feature nets [87]. Arc-labelled FN support a technique for constructing larger feature net from smaller ones to model the addition of new features to an SPL. Section 2.6 discusses advantages and possible future improvements of our approaches. Section 2.7 surveys related work, and Section 2.8 concludes the chapter.

2.2 Software Product Line Modelling Challenge

We illustrate the modelling challenge in software product line engineering using an example of a software product line of coffee vending machines. A

manufacturer of coffee machines offers products to match different demands, from the basic black coffee dispenser to more sophisticated machines, such as ones that can add milk or sugar, or charge a payment for each serving. Each machine variant needs to run software adapted to the selected set of hardware features. Such a family of different software products that share functionality is typically developed using an SPLE approach, that is, as one piece of software structured along distinct features. This approach has major advantages in terms of code reuse, maintenance overhead, and so forth. The challenge is ensuring that the software works appropriately in all product configurations.

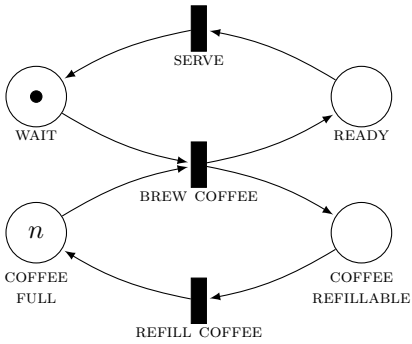


Figure 2.1: Petri net model of a basic coffee machine that can only dispense coffee. Labels on places indicate states of the system; labels on transitions indicate its behaviour.

Petri nets [84] are used to specify how systems behave. Figure 2.1 presents an example of a Petri net for a coffee machine. This has a capacity for n coffee servings; it can brew and dispense coffee, and refill the machine with new coffee supplies. If we now add an optional *Milk* feature, so that the machine can also add milk to a coffee serving, we need to adapt the Petri net, not only to include the functionality of adding milk, but also to be able to control whether or not this feature is present in the resulting software product.

To address the challenge of modelling a software product line with multiple features, which may or may not be included in any given product, we first introduce transition-labelled feature nets. Feature net transitions are annotated with *application conditions* [99], which are propositional formula over features that reflect when the transition is enabled. Later we introduce a variation of feature nets in which application conditions are placed on arcs, rather than transitions, called arc-labelled feature nets.

One advantage of both transition-labelled and arc-labelled feature nets is that they enable the *superposition* of the behaviour of the various products (given

by different feature selections) in the same model.

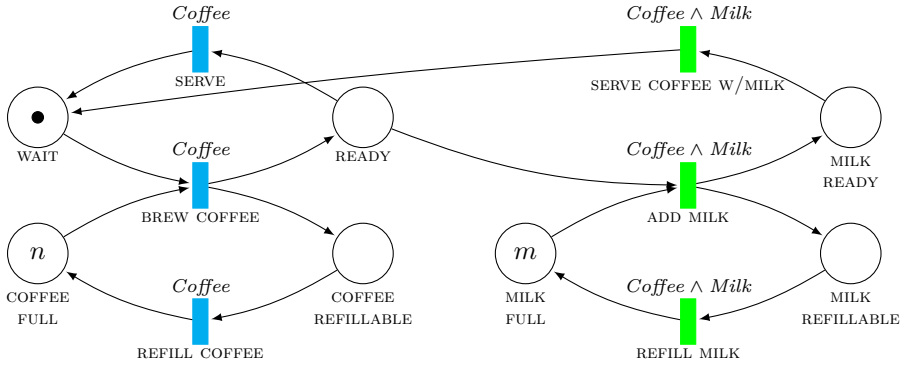


Figure 2.2: Transition-labelled FN of the product line with variants $\{\{Coffee\}, \{Coffee, Milk\}\}$ showing each product in its initial state. Each transition has an application condition attached (label above transitions). Colour is used to visually group transitions according to application conditions.

Figure 2.2 exemplifies a transition-labelled FN of a coffee machine with a milk reservoir. It considers a product line whose products are over the set of features $\{Coffee, Milk\}$, where *Coffee* is compulsory and *Milk* is optional. The conditions on the transitions reflect that the three transitions on the right-hand side can be taken only when both features *Coffee* and *Milk* are present, and the three transitions on the left-hand side can be taken when the *Coffee* feature is present. The restriction of the example net to the transitions that can fire for feature selection $\{Coffee\}$ is exactly the Petri net in Figure 2.1, after removing unreachable places.

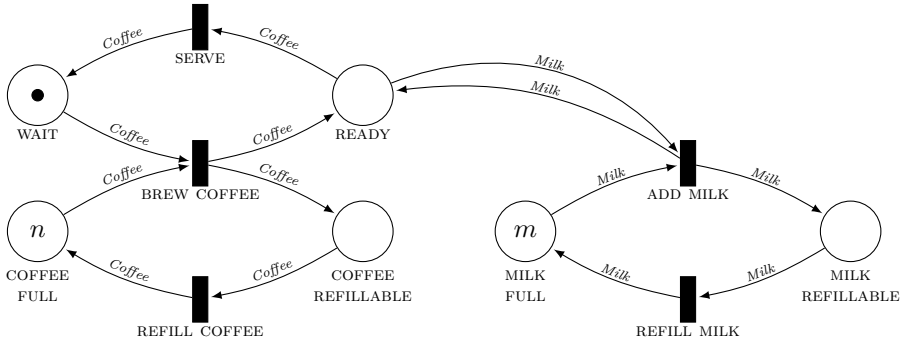


Figure 2.3: Arc-labelled FN of the product line $\{\{Coffee\}, \{Coffee, Milk\}\}$. Each arc has an application condition attached.

Figure 2.3 exemplifies an arc-labelled feature net of the same coffee machine SPL. The application condition above each arc reflects that the arc is present only when the condition evaluates to true. Only then does the arc affect behaviour. If the condition is false, the arc has no effect on behaviour. Consequently, the three transitions on the left-hand side can only fire when the *Coffee* feature is present; the two transitions on the right-hand side can be taken only when the feature *Milk* is present. Observe that the restriction of this example net to the transitions that can fire for feature selection $\{Coffee\}$ is, again, exactly the Petri net in Figure 2.1, after removing unreachable places.

Arc-labelled feature nets have advantages over transition-labelled feature nets when it comes to supporting a modular approach to modelling. This will become clear in Section 2.5.1, where a composition technique for feature nets is proposed.

2.3 Petri Nets

We start with some necessary preliminaries, first by defining multisets and basic operations over multisets. Then we define Petri nets and their behaviour.

Definition 2.3.1 (Multiset)**.** *A* multiset *over a set S is a mapping $M : S \to \mathbb{N}$.*

We view a set S as a multiset in the natural way, that is, $S(x) = 1$ if $x \in S$, and $S(x) = 0$ otherwise. We also lift arithmetic operators to multisets as follows. For any function $\odot : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and multisets M_1, M_2, define $M_1 \odot M_2$ as $(M_1 \odot M_2)(x) = M_1(x) \odot M_2(x)$.

To ground our theory, we recall the terminology and notation surrounding Petri nets [41].

Definition 2.3.2 (Petri Net)**.** *A Petri net is a tuple (S, T, R, M_0), where S and T are two disjoint finite sets, R is a relation on $S \cup T$ (the* flow relation*) such that $R \cap (S \times S) = R \cap (T \times T) = \emptyset$, and M_0 is a multiset over S, called the* initial marking*. The elements of S are called* places *and the elements of T are called* transitions*. Places and transitions are called* nodes*.*

Sometimes we omit the initial marking M_0.

Definition 2.3.3 (Marking of a Petri Net)**.** *A marking M of a Petri net (S, T, R) is a multiset over S. A place $s \in S$ is* marked *iff $M(s) > 0$.*

Definition 2.3.4 (Pre-sets and post-sets)**.** *Given a node x of a Petri net, the set ${}^\bullet x = \{y \mid (y, x) \in R\}$ is the* pre-set *of x and the set $x^\bullet = \{y \mid (x, y) \in R\}$ is the* post-set *of x.*

Definition 2.3.5 (Enabling)**.** *A marking M* enables *a transition $t \in T$ if it marks every place in ${}^\bullet t$, that is, if $M \geq {}^\bullet t$.*

The behaviour of a Petri net is a sequence of states, where each state is defined by a marking. The change from the current state to a new state occurs by the firing of a transition. A transition t can fire if it is enabled. Firing transition t changes the marking of the Petri net by decreasing the marking of each place in the pre-set of t by one, and increasing the marking of each place in the post-set of t by one.

Definition 2.3.6 (Transition occurrence rule)**.** *Given a Petri net $N = (S, T, R)$, a transition $t \in T$* occurs*, leading from a state with marking M_i to a state with marking M_{i+1}, denoted $M_i \xrightarrow{t} M_{i+1}$, iff the following two conditions are met:*

$$M_i \geq {}^\bullet t \qquad \text{(enabling)}$$

$$M_{i+1} = (M_i - {}^\bullet t) + t^\bullet \qquad \text{(computing)}$$

The behaviour defined above is also known as the *firing* of a transition. Transitions fire sequentially, that is, only one transition occurs at a time.

Definition 2.3.7 (Petri net trace)**.** *Given a Petri net $N = (S, T, R, M_0)$, the behaviour the net exhibits by passing through a sequence of states with markings $M_0, \ldots, M_n$, where each change of marking is triggered by a transition occurrence $M_i \xrightarrow{t_i} M_{i+1}$, is called a* trace*. A trace is written $M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} M_n$.*

Definition 2.3.8 (Petri net behaviour)**.** *The behaviour of a Petri net is given by the set of all traces from a given initial marking.*

For example, the following trace is part of the behaviour of the coffee machine Petri net illustrated in Figure 2.1 (the tuples represent markings of the places listed on the left).

$$\begin{array}{r} \text{WAIT} \\ \text{READY} \\ \text{COFFEE FULL} \\ \text{COFFEE REFILLABLE} \end{array} \begin{pmatrix} 1 \\ 0 \\ n \\ 0 \end{pmatrix} \xrightarrow{\text{BREW COFFEE}} \begin{pmatrix} 0 \\ 1 \\ n-1 \\ 1 \end{pmatrix} \xrightarrow{\text{SERVE}} \begin{pmatrix} 1 \\ 0 \\ n-1 \\ 1 \end{pmatrix}$$

2.4 Transition-Labelled Feature Nets

Transition-labelled Feature nets are a Petri net variant used to model the behaviour of an entire software product line. For this purpose, transition-labelled

FN have *application conditions* [99] attached to their transitions. An application condition is a boolean logical formula over a set of features, describing the feature combinations to which the transition applies. It constitutes a necessary (although not sufficient) condition for the transition to fire. In effect, if the application condition is false, it is as if the transition was not present.

Throughout this section, the term feature net (FN) refers to a transition-labelled feature net. We define feature nets and give their semantics. We present two semantic accounts of FN. First, when a set of features is selected, an FN *directly* models the behaviour of the product corresponding to the feature selection. Second, by *projecting* an FN onto a feature selection, one obtains a Petri net describing the behaviour of the same product. We show that these two notions of semantics coincide.

Definition 2.4.1 (Application condition [99])**.** *An* application condition φ *is a propositional formula over a set of features F, defined by the following grammar:*

$$\varphi \quad ::= \quad a \mid \varphi \wedge \varphi \mid \neg\varphi \mid \top,$$

where $a \in F$. The remaining logical connectives can be encoded as usual. Write Φ_F to denote the set of all application conditions over F.

Definition 2.4.2 (Satisfaction of application conditions)**.** *Given an application condition $\varphi \in \Phi_F$ and a set of features $FS \subseteq F$, called a* feature selection*, we say that FS satisfies φ, written as $FS \models \varphi$, defined as follows:*

$$\begin{array}{ll} FS \models \top & \textit{always} \\ FS \models a & \textit{iff } a \in FS \\ FS \models \varphi_1 \wedge \varphi_2 & \textit{iff } FS \models \varphi_1 \textit{ and } FS \models \varphi_2 \\ FS \models \neg\varphi & \textit{iff } FS \not\models \varphi. \end{array}$$

After formally recalling Petri nets and application conditions, we are now in the position to introduce feature nets.

Definition 2.4.3 (feature net)**.** *A feature net is a tuple $N = (S, T, R, M_0, F, f)$, where (S, T, R, M_0) is a Petri net, F is set of features, and $f : T \to \Phi_F$ is a function associating each transition with an application condition from Φ_F.*

For $f(t)$, the application condition associated with transition t, write φ_t. For conciseness, we say that a feature selection FS satisfies transition t whenever $FS \models \varphi_t$.

We now define the behaviour of feature nets for a given (static) feature selection.

Definition 2.4.4 (Transition occurrence rule for FN)**.** *Given a feature net* $N = (S, T, R, M_0, F, f)$ *and a feature selection* $FS \subseteq F$*, a transition* $t \in T$ occurs*, leading from a state with marking* M_i *to a state with marking* M_{i+1}*, denoted* $(M_i, FS) \xrightarrow{t} (M_{i+1}, FS)$*, iff the following three conditions are met:*

$$M_i \geq {}^\bullet t \qquad \text{(enabling)}$$

$$M_{i+1} = (M_i - {}^\bullet t) + t^\bullet \qquad \text{(computing)}$$

$$FS \models \varphi_t \qquad \text{(satisfaction)}$$

In the above definition the state of the Petri net is denoted by a tuple consisting of a marking and a feature selection, even though we assume the feature selection is static (constant). Later on, we will look at dynamic feature selections which can change during execution.

The transition rule for FN is used to define traces that describe the FN's behaviour in the same way as Petri nets.

Definition 2.4.5 (FN Trace)**.** *Given a feature net* $N = (S, T, R, M_0, F, f)$ *and a feature selection* $FS \subseteq F$*, the behaviour the net exhibits by passing through a sequence of markings* $M_0, \ldots, M_n$*, where each change of marking is triggered by a transition occurrence* $(M_i, FS) \xrightarrow{t_i} (M_{i+1}, FS)$*, is called a* trace over FS*. A trace is written* $(M_0, FS) \xrightarrow{t_0} (M_1, FS) \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} (M_n, FS)$*.*

Given an FN, there is a set of traces representing the behaviour of the FN for each feature selection.

Definition 2.4.6 (FN behaviour for a given feature selection)**.** *Given a feature net* $N = (S, T, R, M_0, F, f)$ *and a feature selection* $FS \subseteq F$*, the behaviour of* N *for* FS*, denoted* $\mathrm{Beh}(N, FS)$ *is the set of all traces over* FS *from the initial marking* M_0*.*

If we consider all possible feature selections, we can express the behaviour of the FN.

Definition 2.4.7 (FN Behaviour)**.** *Given a feature net* $N = (S, T, R, M_0, F, f)$*, we define* $\mathrm{Beh}(N)$ *to be the combined set of behaviours for all feature selections over* F*:*

$$\mathrm{Beh}(N) = \bigcup_{FS \in \mathcal{P}F} \mathrm{Beh}(N, FS).$$

2.4.1 Projection-based Semantics of FN

We now present an alternative semantics of feature nets. Given a feature selection, the semantics of an FN is a Petri net consisting of just the transitions satisfying the feature selection.

Definition 2.4.8 (Projection)**.** *Given a feature net $N = (S, T, R, M_0, F, f)$ and a feature selection $FS \subseteq F$, the* projection *of N onto FS, denoted $N{\downarrow}FS$, is a Petri net (S, T', R', M_0), with $T' = \{t \in T \mid FS \models \varphi_t\}$ and the flow relation $R' = R \cap ((S \cup T') \times (S \cup T'))$.*

One *projects* N onto a feature selection FS by evaluating all application conditions φ_t with respect to FS for transitions $t \in T$. If FS does not satisfy φ_t, then transition t is removed from the Petri net. All application conditions are also removed when projecting.

For example, by projecting the FN of the product line $\{\{Coffee\}, \{Coffee, Milk\}\}$ (Figure 2.2) onto the feature selection $\{Coffee\}$, the application condition *Coffee* (on transitions SERVE, BREW COFFEE and REFILL COFFEE) evaluates to true, while the application condition $Coffee \wedge Milk$ (on SERVE COFEEE W/MILK, ADD MILK and REFILL MILK) evaluates to false. Hence, the latter transitions are removed, along with unreachable places. The result is the Petri net depicted in Figure 2.1.

The behaviour of the projection of a Feature Petri net N onto a feature selection FS coincides with the behaviour of N for FS, as stated by the following theorem.

Theorem 2.4.9. *Given a feature net N and $FS \subseteq F$, then:*

$$\mathrm{Beh}(N, FS){\downarrow}FS \;=\; \mathrm{Beh}(N{\downarrow}FS).$$

By projecting $\mathrm{Beh}(N, FS)$ onto the feature selection FS, the feature selection is removed from the traces of N's behaviour.

Proof. ($\subseteq$) We show that every trace $\sigma \in \mathrm{Beh}(N, FS){\downarrow}FS$ is also a trace in $\mathrm{Beh}(N{\downarrow}FS)$. Firstly, the initial markings M_0 coincide in both Petri nets. Secondly, if $(M, FS) \xrightarrow{t} (M', FS)$ then, by Definition 2.4.6, $FS \models \varphi_t$, and by Definition 2.4.8 it is also a transition of $N{\downarrow}FS$. Hence, $M \xrightarrow{t} M'$.

($\supseteq$) Following a similar reasoning as before, we show that every trace $\sigma \in \mathrm{Beh}(N{\downarrow}FS)$ is also a trace in $\mathrm{Beh}(N, FS)$. Observe that, if $M \xrightarrow{t} M'$, then t is a transition of $N \downarrow FS$, and by Definition 2.4.8 $FS \models \varphi_t$. Hence, by Definition 2.4.6 we conclude that also $(M, FS) \xrightarrow{t} (M', FS)$. □

2.4.2 Dynamic Feature Nets

Dynamic Software Product Lines (DSPL) is an area of research concerned with runtime variability of systems [57]. DSPL is an umbrella concept that addresses dynamic reconfiguration of products (i.e. features are added and removed at runtime), but also dynamic evolution of the product line itself (typically referred to as "meta-variability"). Pushing the binding time of features to runtime is often motivated by a changeable operational context, to which a product has to adapt in order to provide context-relevant services or meet quality requirements.

To accommodate modelling the kind of dynamic feature reconfiguration that is characteristic of DSPLs, we introduce Dynamic feature nets (DFN). DFN associate simple update expressions to transitions. Upon firing of a transition, updates affect the feature selection in effect.

Dynamic Feature Reconfiguration Example

Assuming that a product is composed from a static selection of features is sometimes too restrictive. As an example, we can think of a modular appliance, some of whose features can be enabled/disabled temporarily based on the connected hardware modules. For example, a coffee machine using fresh milk instead of milk powder allows the removal of the milk reservoir, in order to store it in the fridge. That change in the hardware configuration may entail a change in the software configuration. Modelling the presence/absence behaviour of the *Milk* feature may entail a significant modelling effort.

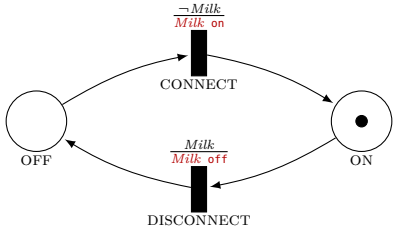


Figure 2.4: DFN modelling the ability to connect/disconnect a feature at runtime.

In our example, switching the *Milk* feature on and off can be modelled by the DFN in Figure 2.4, as an independent addition to the model in Figure 2.2. Associated to the DISCONNECT transition is the *update* expression "`Milk off`". By firing the DISCONNECT transition, the current feature selection is updated, dropping the *Milk* feature. This action globally disables all transitions whose

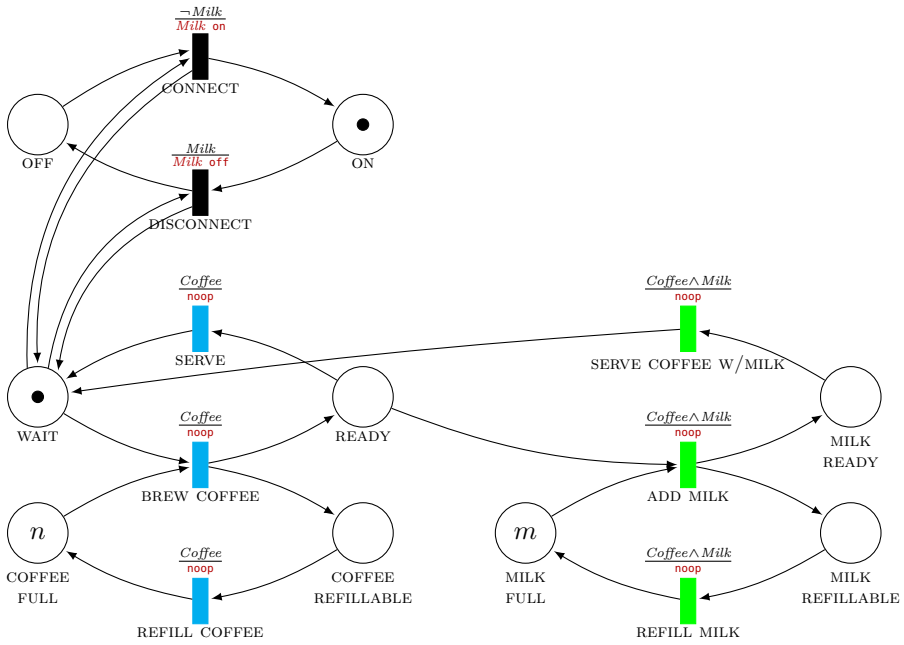


Figure 2.5: DFN (initial state) of a dynamically reconfigurable product line. Whenever transition DISCONNECT fires, feature *Milk* is switched off, disabling all transitions that are conditioned on *Milk*. It is enabled again by firing CONNECT.

application condition depends on the *Milk* feature (that is, ADD MILK, REFILL MILK and SERVE COFFEE W/MILK in Figure 2.2). Conversely, firing the CONNECT transition re-enables all transitions conditioned on the *Milk* feature.

The feature reconfiguration model can remain disconnected from the "functional" model if the user interaction of removing/reconnecting the *Milk* feature can occur independently of the state the coffee machine. Alternatively, we can assume that the reconfiguration of features depends on the functional model. Figure 2.5 shows a model where removing/reconnecting the milk reservoir is only allowed when the machine is in a waiting state, prohibiting, for example, its removal when the machine is in the process of brewing coffee.

Definition

We extend the definition of feature nets to capture the dynamic reconfiguration of products, resulting in a more general Petri net model. In our approach we

associate to each transition an *update expression* that describes how the feature selection evolves after the transition. The resulting model is called *dynamic feature nets* (DFN). DFN extend feature nets by adding a variable feature selection to the state of the Petri net, and associating application conditions and update expressions over the feature set to the transitions. This extension enables a more concise description of SPLs, without adding expressive power with respect to Petri nets (see Section 2.6 for a justification of this claim). We now define update expressions before formalising DFN.

Definition 2.4.10 (Update)**.** *An* update *is defined by the following grammar:*

$$u \quad ::= \quad \texttt{noop} \;\mid\; a\ \texttt{on} \;\mid\; a\ \texttt{off} \;\mid\; u;u$$

where $a \in F$ and F is a set of features. We write U_F to denote the set of all updates over F.

Given a feature selection $FS \in F$, an update expression modifies FS according to the following rules:

$$FS \xrightarrow{\texttt{noop}} FS$$

$$FS \xrightarrow{a\ \texttt{on}} FS \cup \{a\}$$

$$FS \xrightarrow{a\ \texttt{off}} FS \setminus \{a\}$$

$$\frac{FS \xrightarrow{u_0} FS' \qquad FS' \xrightarrow{u_1} FS''}{FS \xrightarrow{u_0;u_1} FS''}$$

Definition 2.4.11 (Dynamic feature net)**.** *A DFN is a tuple $N = (S, T, R, M_0, F, f, u)$, where (S, T, R, M_0, F, f) is an FN and u is a function $T \to U_F$, associating each transition with an update from U_F.*

We write u_t to denote the update expression $u(t)$ associated with a transition t.

Definition 2.4.12 (DFN transition occurrence)**.** *Given a DFN $N = (S, T, R, M_0, F, f, u)$ and an initial feature selection $FS_0 \subseteq F$, a transition $t \in T$* occurs*, leading from a state (M_i, FS_i) to a state (M_{i+1}, FS_{i+1}), denoted $(M_i, FS_i) \xrightarrow{t} (M_{i+1}, FS_{i+1})$, iff the following four conditions are met:*

$$M_i \geq {}^\bullet t \qquad \text{(enabling)}$$

$$M_{i+1} = (M_i - {}^\bullet t) + t^\bullet \qquad \text{(computing)}$$

$$FS_i \models \varphi_t \qquad \text{(satisfaction)}$$

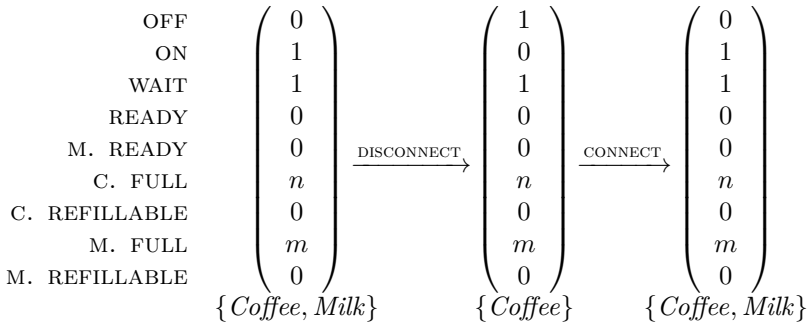
$$FS_i \xrightarrow{u_t} FS_{i+1} \qquad \text{(update)}$$

Definition 2.4.13 (DFN trace)**.** *Given a DFN $N = (S, T, R, M_0, F, f, u)$, the behaviour the net exhibits by assuming a sequence of states $(M_0, FS_0) \dots (M_n, FS_n)$, where each change of state is triggered by a transition occurrence $(M_i, FS_i) \xrightarrow{t_i} (M_{i+1}, FS_{i+1})$, is called a* trace*. A trace is written $(M_0, FS_0) \xrightarrow{t_0} (M_1, FS_1) \xrightarrow{t_1} \dots \xrightarrow{t_{n-1}} (M_n, FS_n)$.*

If we consider all possible traces, we obtain the behaviour of the FN.

Definition 2.4.14 (DFN Behaviour)**.** *Given a DFN $N = (S, T, R, M_0, F, f, u)$, we define* Beh$(N)$ *to be the set of all traces starting with the initial state (M_0, FS_0).*

For example, the following trace is an element of the behaviour of the DFN illustrated in Figure 2.5 (tuples represent markings and the sets below are feature selections).



2.5 Arc-Labelled Feature Nets

A feature net (FN) is a Petri net variant used to model the behaviour of an entire software product line. Arc-labelled feature nets are a FN variant that have application conditions attached to their *arcs*. As defined in Section 2.4, an application condition is a propositional logical formula over a set of features. For arc-labelled feature nets, it describes the feature combinations to which the arc applies. If the application condition is false for a given feature selection, it is as if the arc were not present. Arc-labelled feature nets allow a technique for constructing larger feature nets from smaller ones to model the addition of new features to an SPL. Along with presenting the composition technique, we provide correctness criteria for ensuring that the resulting composition preserves the

behaviour of the original model(s). Arc-labelled feature nets are as expressive as transition-labelled FNs, as will be shown in Section 2.6. The main difference is that they allow a finer grained association with features, which often results in more concise models.

Throughout this section, whenever the term feature net is used, it refers to the arc-labelled variant. We define arc-labelled feature nets and their behaviour by adapting the definition of feature nets described in Section 2.4, where application conditions apply to transitions instead of arcs.

Definition 2.5.1 (Feature Net)**.** *A feature net is a tuple $N = (S, T, R, M_0, F, f)$, where S and T are two disjoint finite sets, R is a relation on $S \cup T$ (the* flow relation*) such that $R \cap (S \times S) = R \cap (T \times T) = \emptyset$, and M_0 is a multiset over S, called the* initial marking*. The elements of S are called* places *and the elements of T are called* transitions*. Places and transitions are called* nodes*. The elements of R are called* arcs*. Finally, F is set of features and $f : R \to \Phi_F$ is a function associating each arc with an application condition from Φ_F. Note that f is different from the function f that associates transitions with application conditions in transition-labelled feature nets (Definition 2.4.3).*

Without f and F, a feature net is just a Petri net. Sometimes we omit the initial marking M_0. The function f determines a node's pre- and post-set, defined below.

Definition 2.5.2 (Marking of a feature net)**.** *A marking M of a feature net (S, T, R, F, f) is a multiset over S. A place $s \in S$ is* marked *iff $M(s) > 0$.*

The pre- and post-sets of arc-labelled FNs depend on the feature selection FS, which determines whether an arc is present or not. The following definition takes this into account. Note that this is different from transition-labelled feature nets, where arcs are fixed.

Definition 2.5.3 (Pre-sets and post-sets)**.** *Given a node x of a feature net and a feature selection FS, the set ${}^{(FS)}x = \{y \mid (y, x) \in R, FS \models f(y, x)\}$ is the* pre-set *of x and the set $x^{(FS)} = \{y \mid (x, y) \in R, FS \models f(x, y)\}$ is the* post-set *of x.*

Definition 2.5.4 (Enabling)**.** *Given a feature selection FS, a marking M* enables *a transition $t \in T$ if it marks every place in ${}^{(FS)}t$, that is, if $M \geq {}^{(FS)}t$.*

We now define the behaviour of feature nets for a given feature selection.

Definition 2.5.5 (Transition occurrence)**.** *Let $N = (S, T, R, M_0, F, f)$ be a feature net and $FS \subseteq F$ a feature selection. A transition $t \in T$* occurs*, leading*

from a state with marking M_i to a state with marking M_{i+1}, denoted $M_i \xrightarrow{t,FS} M_{i+1}$, iff the following two conditions are met:

$$M_i \geq {}^{(FS)}t \qquad \text{(enabling)}$$

$$M_{i+1} = (M_i - {}^{(FS)}t) + t^{(FS)} \qquad \text{(computing)}$$

The transition rule for FN is used to define traces that describe the FN's behaviour. We now define the semantics of a feature net by projecting it onto a Petri net for a given feature selection.

Definition 2.5.6 (Projection)**.** *Given a feature net $N = (S, T, R, M_0, F, f)$ and a feature selection $FS \subseteq F$, the* projection *of N onto FS, denoted $N{\downarrow}FS$, is a Petri net (S, T, R', M_0), with $R' = \{(x, y) \mid (x, y) \in R, FS \models f(x, y)\}$.*

One *projects* N onto a feature selection FS by evaluating all application conditions $f(x, y)$ with respect to FS for all arcs $(x, y) \in R$. If FS does not satisfy $f(x, y)$, then arc (x, y) is removed from the Petri net.

The behaviour of a feature net is the union of the behaviour of the Petri nets obtained by projecting all possible feature selections. The behaviour of a Petri net $N = (S, T, R, M_0)$ is given by the set of all of its traces [49], written $\mathrm{Beh}(N) = \{M_0 \xrightarrow{t_1} \cdots \xrightarrow{t_s} M_n \mid M_i \subseteq S, i \in 1..n, M_{i-1} \xrightarrow{t_i} M_i\}$, and does not include application conditions nor feature selections.

Definition 2.5.7 (FN Behaviour)**.** *Given an FN $N = (S, T, R, M_0, F, f)$, we define* $\mathrm{Beh}(N)$ *as follows:*

$$\mathrm{Beh}(N) = \bigcup_{FS \subseteq F} \mathrm{Beh}(N{\downarrow}FS).$$

2.5.1 Modular Modelling

For a modelling formalism to be useful in practice, it needs to facilitate modular development techniques. This is especially important for modelling software product lines: a single SPL model combines the behaviour of a set of different systems, which are often too complex to develop simultaneously.

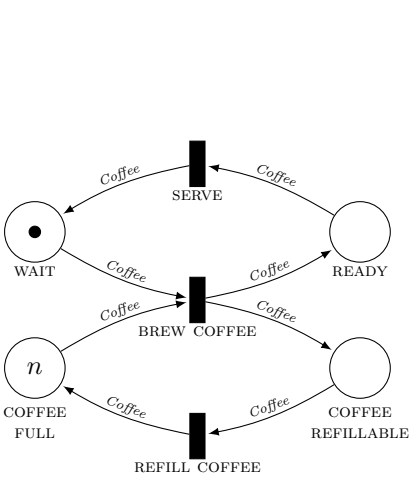
Modular approaches include top-down techniques, where initially an abstract model is sketched and more details are added incrementally, and bottom-up approaches, where subsystems are modelled separately and later plugged together to a global model. Petri nets support both approaches [49]. In the following we propose a bottom-up composition technique for feature nets. It

is based on the idea of modelling features of the system individually and then combining them to obtain a model of the entire SPL. Our approach starts by building a model of the *core* system that is the behaviour which is common to all products of the SPL. Optional features are modelled as separate nets, which also specify how they interact with the core through an *interface*. Core and additional features are then composed stepwise, by incrementally applying each feature to the core. We show how this technique can be applied to modularly specify a coffee machine product line from the three features *Coffee*, *Payment* and *Milk*.

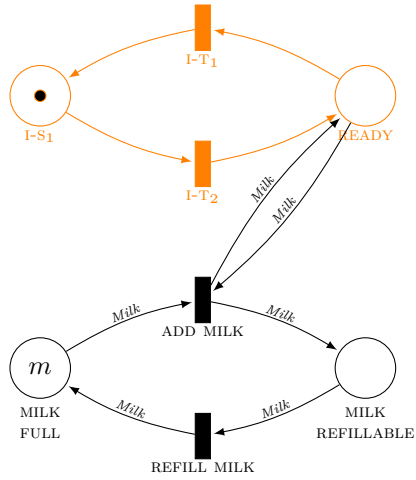
Feature Net Composition

We devise a modular modelling approach in which features (or parts thereof) are first expressed as separate FNs. A feature's interaction with the rest of the system (the *core*) is modelled using an *interface*. Features are modelled separately in such a way that they can be attached to the core, in order to incrementally build a larger model. The interface simulates the behaviour of the core that the features are designed to be plugged into. A feature modelled using this technique can be seen as a partially specified model of the entire SPL, where the feature's behaviour is fully specified, whereas everything else is underspecified. Composition then amounts to connecting the interface to the core to obtain a specification of the combined system. We call a feature net with such an interface a *delta feature net*, as it provides a behavioural delta to the core (i.e., it adds or removes behaviour).

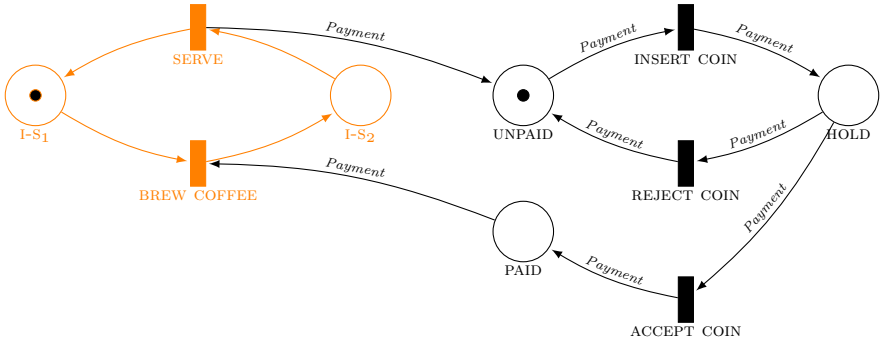
The three features of our example coffee machine are modelled as separate FNs (Figure 2.6). Apart from when a feature's behaviour is self-contained (such as the *Coffee* net in Figure 2.6a) it will typically interact with other features that are part of the larger system. To faithfully model such interactions we include an interface. Interfaces (highlighted in orange in Figure 2.6) abstract the behaviour of the core. The interface will also be used to show that the individual net exposes the same behaviour as it does when it is part of the combined system. For example, the model of *Milk* in Figure 2.6b reflects the fact that adding milk depends on a state of the system in which a cup of fresh coffee is available. The larger system is represented abstractly by the highlighted interface, which models the availability of coffee in the place READY; a token in this place would denote a state in which a freshly brewed cup of coffee is available. Similarly, Figure 2.6c models the fact that after a payment has been accepted, the overall system is able to BREW COFFEE, and after serving the coffee, the system goes back to an UNPAID state. Note that interfaces are in general not limited to the composition with a particular core, but can be attached to any core that they are applicable to.



(a) *Coffee* feature (core)



(b) *Milk* feature



(c) *Payment* feature

Figure 2.6: Individual feature nets modelling the features *Coffee*, *Milk* and *Payment* of a product line of coffee machines. Interfaces are highlighted in orange. Arcs without labels have the application condition *true*.

Constructing a model of the whole SPL is done by stepwise applying the delta nets of each feature to a core model. The intuition behind delta net application is that each interface is replaced with a more complex feature net. In our example, the first step could be to refine *Payment*'s interface by replacing it with the feature net for *Coffee*. In a second step, the feature *Milk* is refined by replacing its interface with the net obtained in the previous step.

We now formally introduce the application of a delta net to a core net.

Definition 2.5.8 (Delta Feature Net)**.** *A delta feature net N is a FN with a designated interface (S_I, T_I), denoted $N = (S, T, R, F, f, S_I, T_I)$, where $S_I \subseteq S$ and $T_I \subseteq T$.*

Delta feature nets specify the behaviour of features designed to be added to a larger system. A sequence of delta FN is combined with a stand-alone FN, the *core*, by sequentially *applying* each delta net to the core. Delta nets include an interface, which models interactions with the core. Such interactions are modelled by transitions or places common to both core and delta net.

Definition 2.5.9 (Delta Net Application)**.** *Let $N = (S, T, R, F, f)$ be a feature net and $D = (S_d, T_d, R_d, F_d, f_d, S_I, T_I)$ a delta feature net with $S \cap S_d \neq \emptyset$. The* application *of D to N results in a net $N' = (S', T', R', F', f')$, written as $N \oplus D$, where*

$$\begin{aligned} S' &= (S_d \setminus S_I) \cup S \\ T' &= (T_d \setminus T_I) \cup T \\ R' &= \{(s,t) \in (R \cup R_d) \mid s \in S', t \in T'\} \\ &\quad \cup \{(t,s) \in (R \cup R_d) \mid t \in T', s \in S'\} \\ F' &= F \cup F_d \\ f' &= f \oslash f_d \end{aligned}$$

and

$$(f \oslash g)(arc) = \begin{cases} f(arc) & \text{if } arc \in \mathrm{dom}(f) \wedge arc \notin \mathrm{dom}(g) \\ g(arc) & \text{if } arc \notin \mathrm{dom}(f) \wedge arc \in \mathrm{dom}(g) \\ f(arc) \wedge g(arc) & \text{if } arc \in \mathrm{dom}(f) \cap \mathrm{dom}(g). \end{cases}$$

When applying a delta net to a core, the interface is dropped and the two nets are "fused" along their common nodes. The arcs that previously connected the delta net interface now connect the core. The applicability of a delta net is limited to certain cores. Let S_B and T_B represent the border of the interface, that is, $S_B = \{s \in S_I \mid \exists t \in T_d \setminus T_I : (s,t) \in R'\}$ and $T_B = \{t \in T_I \mid \exists s \in S_d \setminus S_I : (t,s) \in R'\}$. A delta net is applicable to a core net if the border of the interface is preserved, that is, if $S \cap S_d = S_B$ and $T \cap T_d = T_B$.

We show how delta net application is used to build a model of the example coffee machines SPL. Starting with the separate sub-models in Figure 2.6, delta nets are applied stepwise to a growing core. First, a model with the two features *Coffee* and *Payment* is composed by applying the delta net from Figure 2.6c to the core shown in Figure 2.6a. These nets have the two transitions SERVE and BREW COFFEE in common. The result after applying the delta feature net

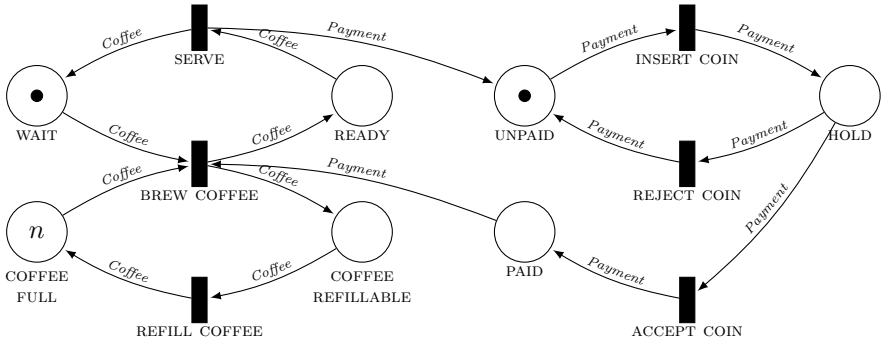


Figure 2.7: A software product line over feature set $\{Coffee, Payment\}$ obtained by applying the delta net *Payment* (Figure 2.6c) to the core net modelling *Coffee* (Figure 2.6a).

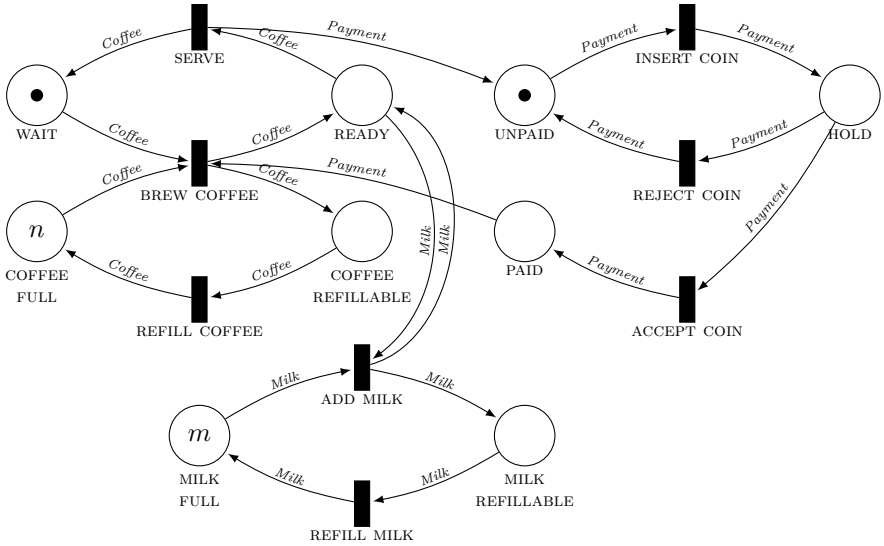


Figure 2.8: FN model of an SPL over the feature set $\{Coffee, Payment, Milk\}$ obtained by sequential application of the delta nets for the features *Payment* (Figure 2.6c) and *Milk* (Figure 2.6b).

is the new core feature net shown in Figure 2.7. In a second step, we add the *Milk* behaviour by applying the feature net in Figure 2.6b to the core obtained in the previous step. These two nets have the place READY in common. The result after delta net application is the model shown in Figure 2.8. Note that the order in which we apply the two delta nets does not matter in this case,

because neither feature (*Milk* or *Payment*) depends on the other. In general, features can depend on other features. This would be reflected by the design of their interfaces, effectively restricting the applicability and ensuring that the delta nets can only be applied in a valid order. As a consequence, delta net application is not commutative.

2.5.2 Behaviour Preservation

When is the application of a delta net D to a core net N *correct*? We consider this application correct if the traces of N and D are in some way the same as the traces of $N \oplus D$, introduced in Definition 2.5.9, after projecting onto the transitions of N and D. However, there are various ways to compare these traces. We can consider only the features used by the original nets (F_N or F_D) or the features used by the combined net ($F_N \cup F_D$). We can consider the core net N or the delta net D. Finally, we can consider the inclusion of traces of the original net in the combined net or also the inclusion of traces of the combined net in the original net. The three dimensions are summarised as:

- **Original** vs. **combined** features. When comparing the behaviour of one of the original nets with the combined net, we can either consider the combined features in the final net or just the features in one of the original nets.
- **Core** vs. **delta**. We can evaluate the correctness of the core or delta net behaviour, always in comparison to the combined net's behaviour.
- **Liveness**, **safety**, or **both**. Preservation of liveness states that a net cannot inhibit behaviour in the other net, while preservation of safety states that a net cannot introduce new behaviour to the other net. For example, we say a delta application is safe with respect to the core net N if the traces of the combined net are included in the traces of N, when considering the common transitions.

By choosing different parameters along these dimensions we obtain different notions of correctness. We formulate a parametrised notion of correctness for the application of delta net D to a core net N as follows:

$$\forall FS \subseteq \Theta_F : \mathrm{Beh}(\Theta_N{\downarrow}FS) \;\; \Theta_R \;\; \mathrm{Beh}((N \oplus D){\downarrow}FS) \quad \text{(param. correctness)}$$

$$\Theta_F \in \{F_N, F_D, F_N \cup F_D\}$$

$$\Theta_N \in \{N, D\}$$

$$\Theta_R \in \{\subseteq, \supseteq, =\}$$

Θ_F can be either the full set of features or the features of the net Θ_N; Θ_N can be either the core or the delta net; and Θ_R is a superset, set inclusion or set equivalence relation between the two sets of traces, with respect to a set of relevant transitions. When Θ_R is a superset relation, it represents *safety*, since no new traces can be introduced by combining the two nets. On the other hand, a subset relation represents *liveness*, since all traces in the original net are still valid traces in the combined net. When we have both safety and liveness assurances, we say that the behaviour is *preserved*, and instantiate Θ_R to be the equality of the traces with respect to the common transitions.

Not all combinations of these dimensions are desirable in all cases. For example, sometimes we might want to inhibit or extend the behaviour of a core net with respect to the combined set of features, breaking the liveness or safety criteria. However, it seems desirable to preserve this behaviour with respect to the features of the core net. In fact, it is open to debate which combination of these dimensions are ideal. We provide sufficient conditions to guarantee:

1. *Preservation* of the behaviour of N with respect to the *original* features ($\Theta_F = F_N$; $\Theta_N = N$; $\Theta_R ==$)
2. *Preservation* of the behaviour of D with respect to the *combined* features ($\Theta_F = F_N \cup F_D$; $\Theta_N = D$; $\Theta_R ==$)
3. *Safety* of the behaviour of N with respect to the *combined* features ($\Theta_F = F_N \cup F_D$; $\Theta_N = N$; $\Theta_R = \supseteq$)

2.5.3 Mathematical Preliminaries

We defined liveness and safety as inclusion of traces with respect to a relevant set of traces. We formalise this concept below.

Definition 2.5.10 (Behaviour inclusion $\subseteq_{Ts}$)**.** *Let $N_i = (S_i, T_i, R_i)$ be a pair of Petri nets, for $i \in 1..2$, and Ts be a set of transitions. We say that the behaviour of N_1 is included by the behaviour of N_2 with respect to Ts, written* $\mathrm{Beh}(N_1) \subseteq_{Ts} \mathrm{Beh}(N_2)$*, if* $\mathrm{Beh}(N_1) \restriction Ts \subseteq \mathrm{Beh}(N_2) \restriction Ts$*, where* $\mathrm{Beh}(N) \restriction Ts = \{tr \restriction Ts \mid tr \in \mathrm{Beh}(N)\}$ *and:*

$$M \restriction Ts = \varepsilon \qquad (M \xrightarrow{t} tr) \restriction Ts = \begin{cases} t \cdot (tr \restriction Ts) & \text{if } t \in Ts \\ tr \restriction Ts & \text{otherwise.} \end{cases}$$

Similarly, we write $\supseteq_{Ts}$ and $=_{Ts}$ to represent superset inclusion and equality for the transitions in Ts.

Behaviour inclusion between two nets N_1 and N_2 is defined by comparing the transition sequences that both nets are able to perform. The transition sequences of both nets are restricted to transitions from a given set T_s. If the transition sequences of N_1 are a subset of the transition sequences of N_2, we say that the behaviour of N_1 is included in the behaviour of N_2.

We now define *weak bisimulation* between two feature nets, which we will use to relate the interface of a delta net with the net to which the delta is applied to, based on the notion of bisimulation described by Schnoebelen and Sidorova [107].

Definition 2.5.11 (Weak bisimulation)**.** *Let $N_i = (S_i, T_i, R_i, M_{0i}, F_i, f_i)$ be two feature nets, for $i \in 1..2$, $\mathcal{M}_i$ the set of markings of N_i, and $\mathcal{B} \subseteq (\mathcal{M}_1 \times \mathcal{M}_2) \cup (T_1 \times T_2)$ a relation over markings and transitions. Recall also the notion of occurrence of transitions introduced in Definition 2.5.5. In the following we write $t \in \mathcal{B}$ to denote that t is in the domain or codomain of $\mathcal{B}$. $\mathcal{B}$ is a weak bisimulation if, for any feature selection FS:*

1. $M_{01}\ \mathcal{B}\ M_{02}$
2. $\forall (M_1, M_2) \in \mathcal{B}$, *if* $M_1 \xrightarrow{t_1, FS} M_1'$ *and* $t_1 \notin \mathcal{B}$, *then* $M_1'\ \mathcal{B}\ M_2$;
3. $\forall (M_1, M_2) \in \mathcal{B}$, *if* $M_1 \xrightarrow{t_1, FS} M_1'$ *and* $t_1 \in \mathcal{B}$, *then there exists* $t_2 \in T_2$ *and* M_2' *such that* $M_2 \xrightarrow{t_2, FS}_{\mathcal{B}} M_2'$, $M_1'\ \mathcal{B}\ M_2'$, *and* $t_1\ \mathcal{B}\ t_2$;
4. *conditions (2) and (3) also hold for* $\mathcal{B}^{-1}$;

where $M \xrightarrow{t, FS}_{\mathcal{B}} M'$ denotes that there are n transitions $t_1 \dots t_n$ such that $M \xrightarrow{t_1, FS} \cdots \xrightarrow{t_n, FS} M_n \xrightarrow{t, FS} M' \quad$ and $\quad \forall j \in 1..n : t_j \notin \mathcal{B}$.
If a weak bisimulation exists between N_1 and N_2 we say that they are weakly bisimilar, written $N_1 \approx N_2$.

Let C be the feature net for the *Coffee* feature (Figure 2.6a), and P the delta net dealing with *Payment* (Figure 2.6c). The interface of P can be seen as a feature net P_I. It holds that $C \approx P_I$. Furthermore, there exists a weak bisimulation $\mathcal{B}$ that relates the transitions with the same name of the two nets, namely SERVE and BREW COFFEE. More specifically, the relation $\mathcal{B}$ below is a bisimulation, where we write $\mathcal{M}_C$ and $\mathcal{M}_{P_I}$ to denote all the markings of C

and P_I, respectively.

$$\{(M, M') \mid M \in \mathcal{M}_C, M' \in \mathcal{M}_{P_i}, M(\textsc{wait}) = 1, M'(\textsc{wait}) = 1\} \cup$$

$$\{(M, M') \mid M \in \mathcal{M}_C, M' \in \mathcal{M}_{P_i}, M(\textsc{wait}) = 0, M'(\textsc{wait}) = 0\} \cup$$

$$\{(\textsc{serve}, \textsc{serve}), (\textsc{brew coffee}, \textsc{brew coffee})\}$$

2.5.4 Preservation of the core behaviour for the original features

Our first criterion compares the core net with the combined net, considering only the features originally present in the core net. If we require the behaviour of the core net to be *preserved* in the combined net, then their traces must coincide with respect to the transitions in the core net. We formalise this criterion as follows.

Criterion 1 (preservation/core/original)**.** *Let $N = (S, T, R, F, M_0, f)$ be a core net and D a delta net. We say that $N \oplus D$ preserves the behaviour of N for the features in F iff*

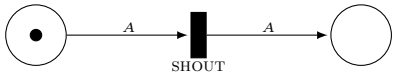
$$\forall FS \subseteq F \ : \ \mathrm{Beh}(N{\downarrow}FS) =_T \mathrm{Beh}(N \oplus D{\downarrow}FS).$$

To verify that a delta net application obeys the above correctness criteria, it is sufficient (although not necessary) to verify the following condition. Check that the arcs between the interface and the non-interface nodes of D require at least one 'new' feature to be present. By new feature we mean a feature that is not in F. This syntactic check ensures that, when considering only the features from the core net, the arcs connecting it to the delta net will never be active.

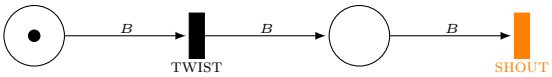
Theorem 2.5.12. *Let $D = (S_d, T_d, R_d, M_{0d}, F_d, f_d, S_I, T_I)$ be a delta net, $N = (S, T, R, M_0, F, f)$ a feature net, and $R_I \subseteq R_d$ be the set of arcs connecting interface nodes $(S_I \cup T_I)$ to non-interface nodes. The behaviour of N is preserved by $N \oplus D$ for the features in F (Criterion 1) if:*

$$\forall (x, y) \in R_I \ : \ \forall FS \in F_d \cup F \ : \ FS \models f(x, y) \quad \mapsto \quad FS \cap (F_d \backslash F) \neq \emptyset. \quad (2.1)$$

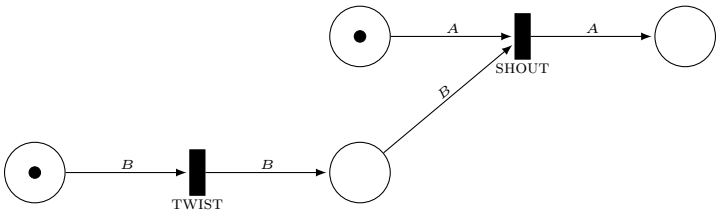
A proof for this theorem can be found in Appendix A.1. In both our examples of delta application, that is, adding payment to a coffee machine and adding milk to the resulting net, the condition in Equation (2.1) holds. The intuition is that, for example, when the *Payment* feature is not available, the *Coffee* feature net is detached from the *Payment* feature net in the combined net. Hence its behaviour is not affected by the *Payment* net and is preserved.



(a) Core net N



(b) Delta net D



(c) $N \oplus D$

Figure 2.9: Example of an FN composition that is correct w.r.t. Criterion 1

Example The following simple example illustrates this criterion. Figure 2.9 shows (a) a core net N with feature set {A}, (b) a delta net D with feature set {B} and (c) the combined net $N \oplus D$ obtained by applying the delta net to the core. Criterion 1 verifies that the net $N \oplus D$ preserves the behaviour of N for the feature selection {A} by using Definition 2.5.10 to compare the behaviour of the two nets $N{\downarrow}\{A\}$ and $N \oplus D{\downarrow}\{A\}$:

$$\begin{aligned} \mathrm{Beh}(N{\downarrow}\{A\}) \restriction \{\textsc{twist}\} &= \{\textsc{twist}\} \\ \mathrm{Beh}(N \oplus D{\downarrow}\{A\}) \restriction \{\textsc{twist}\} &= \{\textsc{twist}\} \\ \Longrightarrow \mathrm{Beh}(N{\downarrow}\{A\}) &=_{\{\textsc{twist}\}} \mathrm{Beh}(N \oplus D{\downarrow}\{A\}). \end{aligned}$$

To check Criterion 1 we can also use Theorem 2.5.12 and simply observe that the application condition on the arc between the interface and the non-interface nodes of D requires the (additional) presence of feature B.

2.5.5 Preservation of the delta behaviour for the combined features

We now define the second correctness criterion.

Criterion 2 (preservation/delta/combined)**.** *Let $N = (S, T, R, M_0, F, f)$ be a core net and $D = (S_d, T_d, R_d, M_{0d}, F_d, f_d, S_I, T_I)$ a delta net. We say that $N \oplus D$ preserves the behaviour of D with respect to features from the combined net iff*

$$\forall FS \subseteq F \cup F_d \ : \ \mathrm{Beh}(D{\downarrow}FS) =_{T_d \setminus T_I} \mathrm{Beh}(N \oplus D{\downarrow}FS).$$

As with the correctness Criterion 1, we present a sufficient condition that guarantees the preservation of the Criterion 2. However, as opposed to the previous case, this condition is based on a *semantic property* of the interface and the core net.

Theorem 2.5.13. *Let $D = (S_d, T_d, R_d, M_{0d}, F_d, f_d, S_I, T_I)$ be a delta net, $N_I = (S_I, T_I, R_I, M_{0D}, F_d, f_d)$ be the interface of D, $N = (S, T, R, F, f)$ a (core) feature net, and $R_B \subseteq R_d$ denote the arcs connecting interface to non-interface nodes. The behaviour of D is preserved by $N \oplus D$ (Criterion 2) if $N \approx N_I$ and there is a specific weak bisimulation $\mathcal{B}$ between N and N_I such that:*

$$\{(t,t) \mid t \in T \cap T_I\} \subseteq \mathcal{B}, \tag{2.2}$$

$$\forall s \in S \cap S_I, (M, M') \in \mathcal{B} : M(s) = M'(s), \tag{2.3}$$

$$\forall (s,t) \in R_B, s \in S_I, (M, M') \in \mathcal{B} \ : \ (M - \{s \mapsto 1\}) \ \mathcal{B} \ (M' - \{s \mapsto 1\}) \tag{2.4}$$

$$\forall (t,s) \in R_B, s \in S_I, (M, M') \in \mathcal{B} \ : \ (M + \{s \mapsto 1\}) \ \mathcal{B} \ (M' + \{s \mapsto 1\}) \tag{2.5}$$

For Equation (2.4) *we assume that, if $M(s) = M'(s) = 0$, then subtracting $\{s \mapsto 1\}$ does not change the markings.*

The proof for Theorem 2.5.13 can be found in Appendix A.2.

Example Recall our running examples. As explained in the end of Section 2.5.3, there is a weak bisimulation between the interface of the delta net for payment P and the core net for coffee C. This simulation obeys Equation (2.2) because the shared transitions are related by $\mathcal{B}$, Equation (2.3) because there places of C and P are disjoint, and Equation (2.4) because, in this case, $\mathrm{dom}(R) \cap S_I = \emptyset$. Hence the composition $CP = C \oplus P$ is correct with respect to Criterion 2.

Consider now the application of the delta net for milk M to the previously obtained core CP. A possible weak bisimulation between CP and the interface of M relates equal markings of the places READY in CP and READY in M, as well as of the places WAIT and I-WAIT. Note that, in order to use Theorem 2.5.13, we need to include markings for any number of tokens in READY, because of Equations (2.4) and (2.5). Furthermore, Equation (2.2) trivially holds, and our specific bisimulation relation $\mathcal{B}$ (which obeys Equations (2.2)–(2.5)) also captures Equation (2.3). We conclude that the composition $CP \oplus M$ is also correct with respect to Criterion 2.

2.5.6 Safety of the core behaviour for the combined features

Our last correctness criterion compares the core net with the combined net with respect to all features, as opposed to the first criterion that only considered the features of the core net. When including the features in the delta net, we consider it *safe* to inhibit traces that were initially possible, provided that no new traces are introduced. We formalise safety using trace inclusion.

Criterion 3 (safety/core/combined)**.** *Let $N = (S, T, R, M_0, F, f)$ be a core net and $D = (S_d, T_d, R_d, M_{0d}, F_d, f_d, S_I, T_I)$ a delta net. We say that $N \oplus D$ is safe with respect to N and to the combined features iff*

$$\forall FS \subseteq F \cup F_d \ : \ \mathrm{Beh}(N{\downarrow}FS) \supseteq_T \mathrm{Beh}(N \oplus D{\downarrow}FS).$$

We claim that, when applying a delta net connecting only places from the interface to the rest of the delta, the delta net application is safe with respect to N and the combined features.

Theorem 2.5.14. *When applying a delta net $D = (S_d, T_d, R_d, M_{0d}, F_d, f_d, S_I, T_I)$ to a core net N, $N \oplus D$ is safe with respect to N and the combined features if:*

$$\forall s \in S_I, t \in T_d \backslash T_I \ : \ (t, s) \notin R_d \quad \wedge \quad \forall t \in T_I, s \in S_d \backslash S_I \ : \ (s, t) \notin R_d. \quad (2.6)$$

The theorem is easily justified by the fact that, after the application, the core net will only be connected to the delta net through arcs pointing from the core to the delta net. These arcs can only further restrict when core transitions can fire.

Example We exemplify the application of two delta nets in this paper: the *Payment* and the *Milk* nets (Figure 2.6c and 2.6b). The first net obeys

Equation (2.6) in Theorem 2.5.14, hence the correctness Criterion 3 holds. The second delta net has arcs connecting places from the interface to a non-interface transition, invalidating Equation (2.6). However, in this case the safety criterion is nevertheless preserved, because a token that exits the core when firing ADD MILK is transported back to its origin in the same step.

2.6 Discussion

Petri nets are a general modelling formalism, proposed for a wide variety of applications. Feature nets and dynamic feature nets leverage the power of Petri nets for modelling static and dynamic software product lines. They combine the behaviour of a set of Petri nets in a single model, thus offering conciseness and convenience when modelling entire software families.

Theorem 2.4.9 shows that a transition-labelled feature net is equivalent in behaviour to a set of Petri nets, one for each product defined by the SPL. Arc-labelled feature nets also do not exceed the expressive power of Petri nets. This is indicated by the fact that an arc-labelled FN can be first encoded as a transition-labelled FN, and then by describing the behaviour of the transition-labelled FN using a set of regular Petri nets. We now justify intuitively this claim.

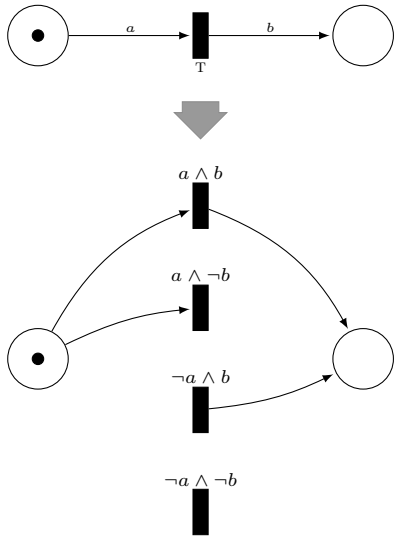


Figure 2.10: Encoding an arc-labelled FN into a transition-labelled FN

The first encoding from arc-labelled FN to transition-labelled FN replaces each transition attached to n arcs in R by 2^n transitions, one for each possible combination of the possible arcs. This is illustrated by an example in Figure 2.10.

The second step, that is, encoding transition-labelled feature nets into Petri nets can be achieved by encoding the satisfaction condition of FN transition occurrences (cf. Definition 2.4.4) by considering for each feature F two places, F ON and F OFF, marked in mutual exclusion depending on whether the feature is selected or not. We illustrate this idea in Figure 2.11. The place MILK ON is associated to the presence of the feature *Milk*. When there is a token in this place, the transitions $\mathrm{T}_1 \ldots \mathrm{T}_n$ are enabled. They are allowed to occur only when there is a token in the place MILK ON. When such a transition occurs this token remains in the same place because MILK ON is part of both the pre- and post-set of the transition. A similar approach can be used to convert any dynamic feature net into a more complex transition-labelled feature net.

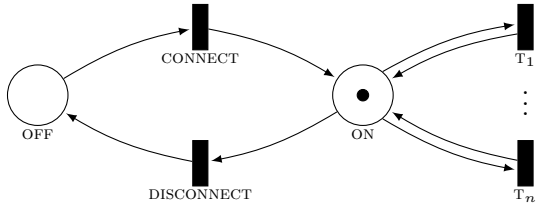


Figure 2.11: Encoding a feature selection as a Petri net marking

Given that feature nets are as expressive as Petri nets, analysis techniques for Petri nets still apply to feature nets. At the same time, feature nets offer a concise way to describe the systems in an SPL.

DFN additionally provision for dynamic SPL, by allowing explicit modelling of feature reconfiguration as part of the behavioural model. By adding update expressions to feature nets, dynamic feature nets do not gain more expressive power than Petri nets, but provide a more elegant separation of concerns. This approach offers orthogonality of the feature reconfiguration from the underlying behaviour, but in a way that enables the reconfiguration to depend upon the underlying behaviour and vice versa.

We present feature nets as a novel SPL modelling formalism, but we do not examine how well this approach fares in practice. If used on a real-world product line, issues of scalability and the practical applicability of our modular modelling workflow could arise. These are subject to future research. In addition, many analysis techniques that exist to determine the behavioural correctness of a Petri net design [84] could be adopted for feature nets. Additionally, these

analyses need to be related and tailored to the domain of SPL engineering [118], which feature nets are designed for.

2.7 Related Work

Our research relates to Petri net based formalisms, behavioural specification of software product lines and dynamic SPL research. We highlight the most relevant works in these areas.

2.7.1 Petri Net Extensions

Petri net composition and decomposition strategies that preserve some properties of the initial net(s) have been studied thoroughly [17, 109, 107, 49].

In *Open Petri Nets* [10], places designated as open represent an interface towards the environment. Open nets are composed by fusing common open places, and the composition operation is shown to preserve behaviour with respect to an inverse decomposition operation. Our Petri net model uses a similar notion of interface, which includes an abstraction of the net that will be matched during application. We use an incremental approach using application of deltas instead of a symmetric composition operation, guided by the intuition that larger systems are build by extending more fundamental systems. The main focus of open Petri nets is the study of properties in a category of nets, while we have a more practical focus on the incremental development of nets.

Inhibitor arc Petri nets [2] can test whether a place is empty by conditioning transitions on the absence of tokens. By modelling individual features as places, the presence or absence of tokens could represent whether a feature is on or off. An application condition could be encoded by including feature places in the pre-sets of transitions, thereby conditioning its firing on the presence or absence of features. Compared to our proposed approach, this entails a more complex net, with unclear boundaries between the functional and structural models.

Conditional Petri nets [37] associate a transition to a formal language over transitions. Extending the classical occurrence rule, a transition is enabled only if the sequence of transitions that occurred in the past is in that language. An FN could be encoded as a conditional Petri net by encoding application conditions in a language over the alphabet of transitions.

In *self-modifying Petri nets* [113], the flow relation changes dynamically according to the number of tokens at certain places in the net. A transition is

enabled if it can fire as many tokens as present in the places referenced by its incoming arcs.

Dynamic Petri nets [48] are similar to self-modifying Petri nets, but have an external control through which the net's structure can be changed by adding or removing arcs between nodes. Certain behaviour can thus be enabled or disabled by integrating or isolating places and transitions. These Petri net designs, although sporting a mechanism of self-modification, are geared towards dynamic changes in throughput, rather that the discrete activation/deactivation of behaviour offered by DFN.

Using *net rewriting systems* [80], dynamic changes in the configuration of a Petri net are described using a rewriting rule that relates places and transitions of the two net configurations to each other. It is conceivable to model a dynamic SPL as a sequence of configurations and a set of rewriting rules which relate each configuration to the next. The DFN approach, however, has the advantage of using a single model, in which each state clearly references a feature selection.

Compared to the surveyed Petri net formalisms, (D)FN semantics are simpler, being closer to the application domain of variability modelling: through application conditions and update expressions they refer directly to the feature model of the SPL.

2.7.2 Behavioural SPL Models

Various formalisms have been adopted for specifying the behaviour of software product lines, with the aim of providing a basis for analysis and verification of such models. A survey of formal methods for software product lines has recently been published [26].

UML activity diagrams have been used to model the behaviour of SPL by superimposing several such diagrams in a single model [38]. Attached to the activity diagram's elements are "presence expressions," which are similar to application conditions. Compared to activity diagrams, Petri nets have a stronger formal foundation, with a larger spectrum of analysis and verification techniques, although, several studies have expressed the semantics of UML diagram using Petri nets (e.g. [45]).

Gruler et al. extended Milner's CCS with a product line variant operator that allows an alternative choice between two processes [54]. The *PL-CCS* calculus includes information about variability: by defining dependencies between features, one can control the set of valid configurations [53].

Variability is often modelled using transition systems enhanced with product-related information. *Modal transition systems* (MTS) [77] allow optional transitions, lending themselves as a tool for modelling a set of behaviours at once [46]. Generalised extended MTS [44] introduce cardinality-based variability operators and propose to use temporal logic formulas to associate related variation points. Asirelli et al. reason about MTS using propositional deontic logic, which is able to express constraints on variable behaviour [7, 8].

Modal I/O automata [76] are a behavioural formalism for describing the variability of components based on MTS and I/O automata. Mechanisms for component composition are provided to support a product line theory. These approaches do not relate behaviour to elements of a structural variability model.

Featured transition systems (FTS) [33, 32] are an extension of labeled transition systems. Similar to feature nets, transitions are explicitly labeled with respect to a feature model, and a feature selection determines the subset of active transitions. In FTS, transitions are mapped to single features. Transition priorities are used to deal with undesired non-determinism when selecting from transitions associated to different features. With application conditions, priorities are no longer required because we can negate the features in other transitions to obtain the same effect. In later work [34], FTS are enhanced with application conditions, which are called feature expressions.

2.7.3 Dynamic SPLs

To the best of our knowledge, dynamic feature nets is the first formal specification and analysis framework for dynamic SPL behaviour. Cordy et al. [35] extend FTS with a function that determines dynamically whether a transition exists. This achieves a goal similar to our update expressions, enabling the modelling of the evolution of adaptive systems as dynamic SPL.

2.8 Summary

This chapter proposes a formal framework for modelling systems with a high degree of variability, addressing an important challenge in software product line engineering. The modelling formalism used is feature nets, a lightweight Petri net extension, of which we present two variants. In transition-labelled FNs, the firing of transitions is conditional on the presence of certain features through *application conditions*. Arc-labelled FNs place application conditions

on the arcs, effectively determining their presence or absence. For arc-labelled FNs we present an approach to composing behavioural models from separately engineered models of individual features. Three correctness criteria for such compositions are also presented.

The Dynamic FN model extends transition-labelled FNs with the ability to express dynamic variability. *Update expressions* associated with DFN transitions make it easy to model changes in the feature selection of a product based on its execution: firing a transition updates the feature configuration in place. To our knowledge, this is the first model to capture both the variable and dynamic aspects of SPL in a single formalism.

Chapter 3

Language Design: Modelling Software Variability with the ABS Language

The HATS methodology is geared towards modelling highly adaptable systems. Such systems have a high degree of variability to accommodate different requirement and deployment scenarios. Modelling and specifying system variability is often a challenge with traditional modelling or specification languages, as they often do not provide the necessary constructs for this task. Dedicated variability modelling languages can be used; their downside is, however, that they do not integrate tightly with the *primary* language, used for specifying core aspects of the software, such as its architecture, class structure, behaviour or formal properties. Hence, a central task within the HATS project was to create a new language—the Abstract Behavioural Specification (ABS) language—that also addresses software variability as a central concern.

ABS has a modular design, with a core and specialised layers that extend the capabilities of the core. The core language is called *Core ABS* [69]. Core ABS is a multi-paradigm language that supports object-oriented, as well as functional programming and has an active object-based concurrency model [105]. Syntax-wise, Core ABS resembles Java. In fact, the syntax of ABS was deliberately designed to be as close as possible to existing programming languages in order to lower the entry barrier to use ABS. Nevertheless ABS is more a modelling than a programming language, because the design of ABS is strongly focused on providing a language that is easy to analyse. High execution performance,

for example, is not a design goal of ABS. Core ABS can be used to model single software systems. Variability modelling is supported through an ABS extension on top of the core. A tutorial introduction to ABS is available [56]. Moreover, a complete description of all ABS features can be found in the ABS reference manual [1].

This chapter documents the variability modelling extension of the ABS language. Section 3.1 provides an overview of the SPL modelling process using the variability modelling constructs of ABS. The following sections then describe each construct in detail. Sections 3.2 and 3.3 describe how ABS is used to model variability in the problem domain using feature modelling and product selection. Section 3.4 details solution space variability modelling using deltas, and its specific implementation in the ABS language. Section 3.5 describes ABS's SPL configuration language that links the problem and solution domain models. Section 3.6 then presents the ABS compiler tool chain with a focus on its usage for configuring and generating executable software products from ABS models of software product lines. Section 3.7 discusses design choices of the ABS variability modelling extensions and the strengths and limitations that result from these. Section 3.8 surveys related work and Section 3.9 concludes the chapter.

The work described in this chapter was carried out in collaboration with Dave Clarke and José Proença and was published as *Variability Modelling in the ABS Language* at the 10th International Symposium on Formal Methods for Components and Objects (FMCO 2011) [29]. The author of this dissertation contributed mainly towards the integration of the variability language constructs with the ABS language, the extension of the ABS modules system to incorporate deltas and the implementation as part of the ABS tool chain (ABS compiler front-end, back-end and IDE).

3.1 Software Variability Modelling Overview

ABS provides the following language constructs for modelling variable systems following SPL engineering practices [94].

Micro Textual Variability Language (μTVL) is used to model all products of an SPL by using features and feature attributes. A μTVL model hence describes a feature model.

Product selection identifies individual products that are of particular interest for the user. These are defined in terms of a valid combination of features.

Core consists of the ABS classes that implement a core set of functionality of the corresponding product line. Typically, the core represents one product configuration, from which other products are derived through the application of deltas.

Deltas (delta modules) are reusable units of ABS code which can be applied incrementally to the core to adapt its behaviour. They effectively describe how to change the core.

SPL configuration links features to deltas. It ensures that software products can be generated automatically based on the set of features that they implement.

We explain these elements in more detail. As a running examples we use a product line of chat (instant messaging) applications. The Chat SPL describes a set of products that offer a variable set of communication methods such as text, voice and video.

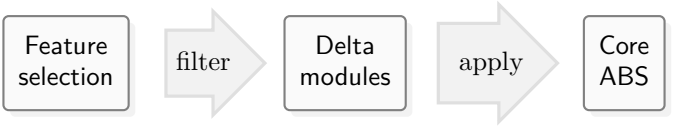


Figure 3.1: Generation of a software product

Figure 3.1 depicts the main steps required to generate a software product using ABS. The developer first selects the desired features. This selection is then used to choose the relevant delta modules. These deltas are applied in a particular sequential order to the core model. The application of all relevant deltas results in a software product with the desired features.

3.1.1 Feature Model

Feature models are an approach to modelling problem space variability. The variable artifacts are represented as features and feature attributes. The relationships among these are described using logical constraints on the combination of features and attributes. Constraints effectively restrict the set of valid products.

Figure 3.2 shows a graphical representation of our example Chat product line feature model using feature diagram notation [72]. The hierarchical organisation ensures that, for a valid product, at least one `Mode` feature is selected; the `Files` feature is optional. The two logical implications below the diagram capture

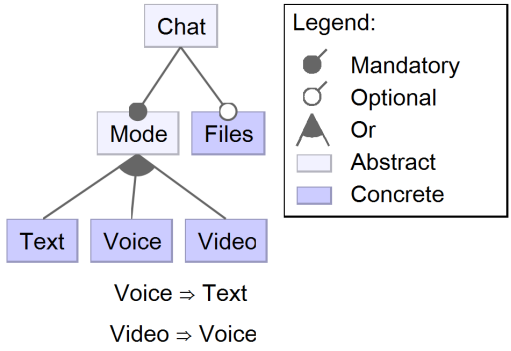


Figure 3.2: Feature diagram of the chat product line

extra constraints, namely that the `Voice` feature requires `Text` to be selected, and `Video` requires `Voice`.

```
root Chat {
   group allof {
      Mode {
         group [1..*] {
            Text, Voice {require: Text}, Video {require: Voice}
         }
      },
      opt Files
   }
}
```

Figure 3.3: μTVL feature model of the chat product line

In ABS feature models are described using μTVL, the micro Textual Variability Language [29], an extended subset of TVL [30]. The advantage over auxiliary modelling notations, such as feature diagrams, is that μTVL models are an integral part of the overall SPL model and can be analysed in connection with the other elements of the SPL. The μTVL code for the Chat product line is presented in Figure 3.3. The μTVL language can also express feature attributes and arbitrary constraints over boolean and integer values. Feature modelling with ABS is explained in more detail in Section 3.2.

3.1.2 Product Selection

A product selection identifies individual products that are of particular interest to the user. Products of static SPLs are selected in ABS simply by associating a product name with a set of features and specifying attributes. In our example, Figure 3.4 declares three products: HighEnd, Regular, and LowEnd, each based on a feature combination that is valid with respect to the feature model (Figure 3.3).

```
product LowEnd (Text);
product Regular (Voice, Text);
product HighEnd (Video, Voice, Text, Files);
```

Figure 3.4: A product selection of the chat product line

3.1.3 Core

The core defines the set of classes, interfaces, functions, etc. that form a base product. In our case, as shown in Figure 3.5, the core module is composed of the classes that are needed to construct a LowEnd chat product (that only supports text-based communication, as defined in the product selection in Figure 3.4).

```
module Chat;
interface Client { ... }
interface Text extends Client {
   Unit message(Client client, String msg);
}
class ClientImpl implements Client, Text {
   Unit message(Client client, String msg) { ... }
}
```

Figure 3.5: The *core* of the chat SPL

3.1.4 Deltas

ABS deltas implement the delta-oriented programming paradigm [101], a software development approach in which program variants are derived from a core program by applying a set of program transformations called deltas. Deltas express the addition, removal, or replacement of program elements such as classes, interfaces, functions, methods, and fields. The details of delta-oriented

programming are presented in Section 3.4. Our example in Figure 3.6 shows a delta `DVoice` that introduces new classes and interfaces, and modifies the core implementation of the class `ClientImpl`.

```
delta DVoice; // modify core to add voice functionality
uses Chat;
adds interface Voice extends Client {
   Call call(Client client);
}
modifies class ClientImpl adds Voice {
   adds Call call(Client c) { ... }
}
adds interface Call { ... }
adds class CallImpl implements Call { ... }
```

Figure 3.6: Definition of deltas for the chat SPL

3.1.5 SPL Configuration

Configuration of a product at compile-time always starts with a core and applies a sequence of deltas to that core. In ABS this sequence is determined by the *SPL configuration*. An SPL configuration links a feature model, which describes the structure of an SPL, to deltas that implement its behaviour. Features and deltas are associated through *application conditions* [99], which are logical expressions over the set of features and attributes in a feature model. The collection of applicable deltas is given by the application conditions that are true for a particular feature and attribute selection. Otherwise stated, if given a product (i.e., a set of features), the application conditions defined for each delta determine whether that particular delta should be applied to the core in order to generate the given software product.

```
productline ChatPL;
features Text, Voice, Video, Files;
delta DVoice when Voice;
delta DVideo when Video;
delta DFiles when Files;
```

Figure 3.7: Configuration of the chat product line

Figure 3.7 shows the product line configuration for the Chat SPL. The two deltas `DVoice` and `DVideo` are associated, respectively, with the application

conditions `Voice` and `Video`. This configuration can be read as: apply the delta `DVideo` whenever the `Video` feature is selected, and apply delta `DVoice` whenever feature `Voice` is selected. More complex application conditions include conjunctions (`a && b`) and negations (`~a`). Furthermore, the SPL configuration can adjust the order in which deltas are applied. For example, the code **delta** `B` **when** `F` **after** `A` declares that, when both deltas `A` and `B` need to be applied, `A` always precedes `B`. The SPL configuration language is documented in more exhaustive detail in Section 3.5.

3.1.6 Interplay of Variability Modelling Constructs

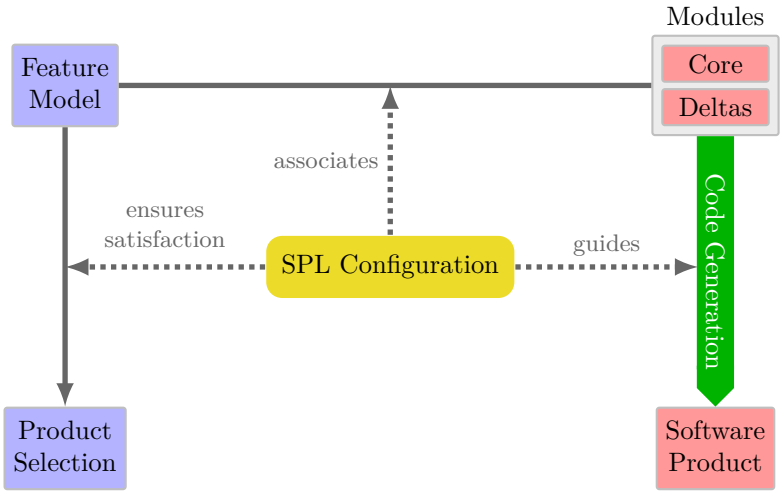


Figure 3.8: ABS variability modelling framework overview

Figure 3.8 shows how the variability modelling constructs of ABS fit together. In the problem space, a feature model describes the variability of the SPL in terms of user-identifiable features and attributes. Products are selected based on the feature model. Over in the solution space, code artifacts are encapsulated as core ABS modules and a set of deltas. Software products are generated by sequentially applying deltas to the core. The SPL configuration ties these elements together. It ensures that the selected products satisfy the feature model; it associates deltas to features; and it guides the generation of software products by ordering the application of deltas.

3.2 Feature Modelling

This section describes in more detail the μTVL text-based feature modelling language, an extended subset of TVL [21, 30]. TVL was developed at the University of Namur, Belgium, to serve as a reference language for specifying feature models. It is textual, as opposed to diagrammatic, and aims to be scalable, concise, modular, and comprehensive, and thus, serves as a suitable starting point for our purposes. A feature model is represented textually as a tree of nested features, each with a collection of boolean or integer attributes. Additional cross-tree dependencies can also be expressed in the feature model.

μTVL is designed to be deliberately smaller than TVL in order to capture the essential feature modelling requirements and to simplify the manipulation of feature models. The simplification allows reducing a number of semantic constraints imposed by TVL to syntactic constraints. μTVL enables a feature model with multiple roots (hence, multiple trees) to express orthogonal variability [94], which is useful for expressing application models and platform models in an orthogonal fashion (even in different files). Support for attributes of enumerated types have been dropped, but our tools support checking of satisfiability of integer attributes. Finally, in μTVL features can only be extended (in *FeatureExtension* clauses) by adding new constraints, but not by introducing new features. Even though TVL syntax is used (with a few variations), the tools for μTVL have been developed from scratch and integrated with the ABS language tool suite.

3.2.1 Syntax

The grammar of μTVL is given in Figure 3.9. Text in `monospace` denotes terminal symbols. Assume the presence of two global sets: FID of feature names and AID of attribute names.

Attributes and values in μTVL range either over integers or booleans. The *Model* clause specifies a number of 'orthogonal' root feature models along with a number of extensions that specify additional constraints, typically cross-tree dependencies. The *FeatureDecl* clause specifies the details of a given feature, firstly by giving it a name (FID), followed by a number of possibly optional sub-features, the feature's attributes and any relevant constraints. The *FeatureExtension* clause specifies additional constraints and attributes for a feature. This is particularly useful for specifying constraints that do not fit into the tree structure given by the root feature model. The *Group* clause specifies the sub-features of a feature. This consists of a specification of the cardinality of the group, plus a number of possibly optional sub-features. The *Cardinality*

```
Model          ::= (root FeatureDecl)* FeatureExtension*

FeatureDecl    ::= FID [{ [Group] AttributeDecl* Constraint* }]
FeatureExtension ::= extension FID { AttributeDecl* Constraint*}

Group          ::= group Cardinality
                   { [opt] FeatureDecl, ([opt] FeatureDecl)* }
Cardinality    ::= allof | oneof | [n1 .. *] | [n1 .. n2]
AttributeDecl  ::= Int AID ;
                 | Int AID in [ Limit .. Limit ] ;
                 | Bool AID ;
Limit          ::= n | *

Constraint     ::= Expr ; | ifin: Expr ; | ifout: Expr ;
                 | require: FID ; | exclude: FID ;
Expr           ::= True | False | n | FID | AID | FID.AID
                 | UnOp Expr | Expr BinOp Expr | ( Expr )
UnOp           ::= ~ | -
BinOp          ::= || | && | -> | <-> | == | != | > | < | >= | <=
                 | + | - | * | / | %
```

Figure 3.9: Grammar of μTVL, the feature modelling language of ABS

clause describes the number of elements of a group that may appear in a result. The *AttributeDecl* clause specifies the declaration of both integer (bounded or unbounded) and boolean attributes of features.

The *Constraint* clause specifies constraints on the presence of features and on attributes. An `ifin` constraint is only applicable if the current feature is selected. Similarly, an `ifout` constraint is only applicable if the current feature is not selected. A `require` clause specifies that the current feature requires some other feature, whereas `exclude` expresses the mutual incompatibility between the current feature and some other feature. The *Expr* clause expresses a boolean constraint over the presence of features and attributes, using standard boolean and arithmetic operators. Features are referred to by identity (FID). Attributes are referred to either using an unqualified name (AID), for in scope attributes, or using a qualified name (FID.AID) for attributes of other features.

Example

Figure 3.10 shows a feature model of a multi-lingual "Hello World" product line, which describes software that can output "Hello World" in multiple languages some number of times. This SPL has two main features, *Language* and *Repeat*,

```
root MultiLingualHelloWorld {
   group allof {
      Language {
         group oneof { English, Dutch, German }
      },
      opt Repeat {
         Int times in [0..1000];
         ifin: times > 0;
      }
   }
}
extension English {
   ifin: Repeat -> (Repeat.times >= 2 && Repeat.times <= 5);
}
```

Figure 3.10: Feature model of a multi-lingual "Hello World" SPL

under the root feature and joined with the **allof** combinator. The *Language* feature requires one out of three possible features: *English*, *Dutch*, or *German*. The *Repeat* feature is optional, it has no associated sub-features, and it has an attribute *times* which ranges between 0 and 1000, with an added condition that it must be strictly greater than 0. An extension for the *English* feature is also given. When the *English* and the *Repeat* features are present, the attribute `times` must be between 2 and 5, inclusive.

3.2.2 Semantics

The semantics of a feature model in μTVL are defined by translation into constraints over integers whose solutions correspond to valid feature and attribute selections. Boolean variables are treated as integers in the standard manner: 0 corresponds to false, and 1 to true. The function $[\![\ ]\!]$ encoding feature model M as an integer constraint is given in Figure 3.11. The notation $\overline{x}$ represents a sequence of elements $x_1 \cdots x_n$. Within the context of a given feature f, function $[\![\ ]\!]_f$ translates constraints relative to that feature. In the translation, $f^\dagger$ is a unique name based on name f. If f is an optional feature, $f^\dagger$ can freely be set to 1 to count the optional feature, even when f is absent. This is used, for example, when dealing with an **allof** constraint, which requires that all children are present; some may however be optional, so as far as the **allof** constraint is concerned, optional children are counted, though the corresponding features may not be included. Expressions e are encoded into constraints, denoted ϕ_e. Their encoding is straightforward and therefore omitted (cf. Classen et al. [30]).

Boolean operations are mapped to a conjunction of integer operations over the values 0 and 1 where, for example, $a \rightarrow b$ is a shorthand for $a \leq b$. Finally, we assume a lower bound MIN and an upper bound MAX on the values of integer variables.

Given a feature model FM in μTVL, the set of solutions of the integer constraints $[\![FM]\!]$ provides our semantics for FM. Such a solution will specify values for all attributes even when the corresponding feature is not selected. Such assignments should be ignored.

The semantics also enforce that each feature is selected either zero or one times, in spite of cardinality conditions which may appear to allow more instances of a feature. Cardinality conditions specify the number of selected sub-features from a group. Note that optional features can only appear under the **allof** cardinality; otherwise there would be a fragile interaction between cardinality conditions and optional features [13].

Figure 3.12 shows the encoding into integer constraints of the Hello World feature model introduced in Figure 3.10. Every declaration of a new feature or attribute x is converted into a constraint of type $min \leq x \leq max$, where in the case of booleans and feature names, $min = 0$ and $max = 1$. The tree structure of the feature model is captured by implications between the children and their parents, as shown in the second and third lines of Figure 3.12. The optional feature `Repeat` is split into two variables: `Repeat` and $\texttt{Repeat}^{\dagger}$. The latter is used only to address the cardinality of the parent `MultiLingualHelloWorld`, and they are connected by the implication $\texttt{Repeat} \rightarrow \texttt{Repeat}^{\dagger}$, similar to how child features are related to their parent. Cardinalities are encoded as constraints that add the 0/1/integer value of the feature variables and check whether they belong to a specific domain, as shown in the third and seventh line of the example. Constraints over attributes are simply interpreted as integer constraints.

3.3 Product Selection

To generate a product from a product line, it is necessary to select a set of features. The product selection language allows the specification of products by selecting their features. A product specified in this manner states which features are to be included in the product and sets the attributes of those features to concrete values. As depicted in Figure 3.8, a product selection is checked against a μTVL feature model for validity. It is then used by the SPL configuration (Section 3.5) to guide the selection and application of deltas (Section 3.4) during the generation of the final software product.

$$
\begin{aligned}
[\![\overline{F}]\!] &= \textstyle\bigwedge_{x\in\overline{F}} [\![x]\!] \\
[\![f\ [G]\ \overline{A}\ \overline{C}]\!] &= (0 \le f \le 1) \wedge [\![[G]\,]\!]_f \wedge [\![\overline{A}]\!] \wedge [\![\overline{C}]\!]_f \\
[\![\texttt{allof}\ \overline{N}]\!]_f &= \mathit{tree}(f, \overline{N}) \wedge \textstyle\sum \overline{N} = \#\overline{N} \wedge [\![\overline{N}]\!] \\
[\![(\texttt{min}\ n)\ \overline{N}]\!]_f &= \mathit{tree}(f, \overline{N}) \wedge n \le \textstyle\sum \overline{N} \wedge [\![\overline{N}]\!] \\
[\![(\texttt{rng}\ n_1\ n_2)\ \overline{N}]\!]_f &= \mathit{tree}(f, \overline{N}) \wedge n_1 \le \textstyle\sum \overline{N} \le n_2 \wedge [\![\overline{N}]\!] \\
[\![\texttt{opt}\ (f\ [G]\ \overline{A}\ \overline{C})]\!] &= f \to f^\dagger\ \wedge\ [\![f\ [G]\ \overline{A}\ \overline{C}\,]\!] \\
[\![\texttt{mand}\ F]\!] &= [\![F]\!] \\
[\![f.a\ \texttt{int}\ L_1\ L_2]\!] &= \mathit{val}_{min}(L_1) \le f.a \wedge \\
&\qquad f.a \le \mathit{val}_{max}(L_2) \\
[\![f.a\ \texttt{bool}]\!] &= 0 \le f.a \le 1 \\
[\![e]\!] &= \phi_e
\end{aligned}
$$

$$
[\![\ [X]\ ]\!] = \begin{cases} [\![X]\!] & \text{if } X \text{ is present} \\ \mathit{true} & \text{otherwise} \end{cases}
$$

$$
\begin{aligned}
\#(N_1 \cdots N_n) &= n \\
\textstyle\sum(N_1 \cdots N_n) &= \mathit{feat}(N_1) + \cdots + \mathit{feat}(N_n) \\
\mathit{tree}(f, N_1 \cdots N_n) &= \textstyle\bigwedge_{1\le i\le n}\ \mathit{feat}(N_i) \to f
\end{aligned}
$$

$$
\begin{aligned}
[\![\texttt{ifin}\ e]\!]_f &= f \to [\![e]\!] \\
[\![\texttt{ifout}\ e]\!]_f &= \neg f \to [\![e]\!] \\
[\![\texttt{require}\ f']\!]_f &= f \to f' \\
[\![\texttt{exclude}\ f']\!]_f &= \neg(f \wedge f') \\
\mathit{feat}(\texttt{opt}(f\ \_\ \_\ \_)) &= f^\dagger \\
\mathit{feat}(\texttt{mand}(f\ \_\ \_\ \_)) &= f \\
\mathit{val}_x(n) &= n \\
\mathit{val}_{min}(*) &= \mathit{MIN} \\
\mathit{val}_{max}(*) &= \mathit{MAX}
\end{aligned}
$$

Figure 3.11: Semantics of μTVL

$$
\begin{aligned}
&0 \leq \texttt{MultiLingualHelloWorld} \leq 1 \ \wedge \\
&\texttt{Language} \rightarrow \texttt{MultiLingualHelloWorld} \wedge \\
&\texttt{Repeat}^{\dagger} \rightarrow \texttt{MultiLingualHelloWorld} \ \wedge \\
&\texttt{Language} + \texttt{Repeat}^{\dagger} = 2 \ \wedge \ 0 \leq \texttt{Language} \leq 1 \ \wedge \\
&\texttt{English} \rightarrow \texttt{Language} \ \wedge \ \texttt{Dutch} \rightarrow \texttt{Language} \ \wedge \ \texttt{German} \rightarrow \texttt{Language} \ \wedge \\
&1 \leq \texttt{English} + \texttt{Dutch} + \texttt{German} \leq 1 \ \wedge \\
&0 \leq \texttt{English} \leq 1 \ \wedge \ 0 \leq \texttt{Dutch} \leq 1 \ \wedge \ 0 \leq \texttt{German} \leq 1 \ \wedge \\
&0 \leq \texttt{Repeat}^{\dagger} \leq 1 \ \wedge \ \texttt{Repeat} \rightarrow \texttt{Repeat}^{\dagger} \ \wedge \\
&0 \leq \texttt{Repeat} \leq 1 \ \wedge \ 0 \leq \texttt{Repeat.times} \leq 1000 \wedge \texttt{Repeat.times} > 0 \ \wedge \\
&\texttt{English} \rightarrow (\texttt{Repeat} \rightarrow (\texttt{Repeat.times} \geq 2 \wedge \texttt{Repeat.times} \leq 5)).
\end{aligned}
$$

Figure 3.12: Semantics of "Hello World" feature model

3.3.1 Syntax

Selection	::=	`product` *TypeId* `(` *FeatureSpecs* `) ;`
FeatureSpecs	::=	*FeatureSpec* (`,` *FeatureSpec*)*
FeatureSpec	::=	FID [*AttributeAssignments*]
AttributeAssignments	::=	`{` *AttributeAssignment* (`,` *AttributeAssignment*)* `}`
AttributeAssignment	::=	AID `=` *Literal*

Figure 3.13: Product selection grammar

Figure 3.13 specifies the grammar of the ABS product selection language. The *Selection* clause specifies a product by giving it a name and by stating the features and optional attribute assignments that are included in that product. The *FeatureSpec* clause specifies that a given feature is present, and the optional *AttributeAssignment* clause is used for specifying the attributes, which are assigned concrete values.

```
product P1 (English);
product P2 (German);
product P3 (German, Repeat{times=10});
product P4 (English, Repeat{times=6}); // should be refused because times>5
```

Figure 3.14: Product selection for the "Hello World" SPL

In the example in Figure 3.14 we specify four products: `P1`, `P2`, `P3`, and `P4`. In the case of the product `P1`, the parameter `English` means the product consists of this feature and of the features implied by the constraints over the feature model. In this case the implied features are `Language` and the root `MultiLingualHelloWorld`, according to the model in Figure 3.10. In `P3` and `P4` the parameters also include attribute values; in these cases a value is assigned to the attribute `times` of the feature `Repeat`.

3.3.2 Semantics

The component of interest in a product selection such as

```
product P (Feature1 {attribute1_1 = value1_1, ...},
        Feature2 {attribute2_1 = value2_1, ...}, ...);
```

is an assigment $\sigma \in \mathit{ProductSelection}$ defined as follows:

- for each $\texttt{Feature}_i$, $\sigma(\texttt{Feature}_i) = 1$.
- for each $\texttt{attribute}_{i,j} = \texttt{value}_{i,j}$ clause in $\texttt{Feature}_i$, $\sigma(\texttt{Feature}_i.\texttt{attribute}_{i,j}) = \texttt{value}_{i,j}$.

The assignment is not complete as it does not specify the values for unselected or implicitly-selected features. An example of an implicitly-selected feature occurs when a leaf feature is selected, requiring that its ancestors in the tree need to be selected too. In addition, the variable $f^\dagger$ introduced to count optional feature f is set to 1. Finally, values of attributes for unselected features are set to some arbitrary value so that the all variables appearing in a constraint are defined (required to test satisfaction). The following steps add the missing elements to an assignment. We call this the *completion* of the product selection. Assume that $f \in \mathsf{FID}$, $a \in \mathsf{AID}$, and feature model FM is encoded as constraints given by $\psi = [\![FM]\!]$.

1. Iterate the following steps until a fixed point is reached:
   (a) If $f \in dom(\sigma)$ and f' is the parent of f, then set $\sigma(f') = 1$.
   (b) If $f \in dom(\sigma)$ and $f^\dagger$ appears in ψ, then set $\sigma(f^\dagger) = 1$.
2. If $f \notin dom(\sigma)$ and f appears in ψ, then set $\sigma(f) = 0$
3. If $f.a \notin dom(\sigma)$ and $f.a$ appears in ψ, then set $\sigma(f.a) = v$, where v is an arbitrary (integer) value within the range specified for $f.a$.

A product selection σ is *valid* whenever all completions σ' are solutions for the feature model FM encoded as ψ, written as $\sigma' \models \psi$.

The product `P3` from the example in Figure 3.14 leads to the following initial variable assignment in the context of the feature model in Figure 3.12.

$$\sigma(\texttt{German}) = 1 \qquad \sigma(\texttt{Repeat}) = 1 \qquad \sigma(\texttt{Repeat.times}) = 10.$$

The remaining variables are `English`, `Dutch`, and `MultiLingualHelloWorld`, which is the parent of `Language` and `Repeat`, and there are no other attributes. The completion of σ includes the following additional elements:

$$\sigma(\texttt{MultiLingualHelloWorld}) = 1 \qquad \sigma(\texttt{English}) = 0$$

$$\sigma(\texttt{Language}) = 1 \qquad \sigma(\texttt{Dutch}) = 0$$

The resulting completed assignment σ satisfies the constraints specified in the example in Figure 3.11. In contrast to this, the constraints would not be satisfied for product `P4`, where $\sigma(\texttt{English}) = 1$, $\sigma(\texttt{Repeat.times}) = 6$, and $\sigma(\texttt{Repeat}) = 1$, due to the clause $\texttt{English} \rightarrow (\texttt{Repeat} \rightarrow (\texttt{Repeat.times} \geq 2 \wedge \texttt{Repeat.times} \leq 5))$.

3.4 Delta Modelling

Delta-oriented programming was introduced by Schaefer et al. [101, 102, 100] as a novel programming language approach for software-based product lines, and as an direct alternative to feature-oriented programming [11]. Both approaches aim at automatically generating software products for a given feature selection by providing a flexible and modular technique to build different products that share common code. In feature-oriented programming, software modules are associated to features, and product generation consists of composing the modules for a feature selection. In delta-oriented programming [101], *application conditions* over the set of features and their attributes, are associated with modules of program modifications (add, remove or modify code), called delta modules. The collection of applicable delta modules is given by the application conditions that are true for a particular feature and attribute selection. By not associating the delta modules directly with features, a degree of flexibility is obtained, resulting in better reuse of code and the ability to resolve conflicts caused by deltas modifying the code base in incompatible ways [28]. The flexibility offers benefits for managing the evolution of product lines, by allowing versions to be implemented using software deltas.

The implementation of a software product line in delta-oriented programming [101] is divided into a *core module* and a set of *delta modules*. The

core module typically consists of the classes that implement functionality common to all products of the corresponding product line. Delta modules describe how to change the core module to obtain new products. The choice of which delta modules to apply is based on the selection of desired features for the final product. Schaefer et al. described and implemented delta-oriented programming for Java [101], introducing the programming language DeltaJava. This language has strongly influenced our design, though we further separate deltas from features by moving application conditions out of deltas and into a product line configuration language. Our approach has been also pursued by Schaefer et al. [102, 100].

Delta modelling is included in the ABS language as part of its SPL development facility and is used to implement variability at the source code level of abstraction. Deltas are applied to the core program by the ABS compiler front end, which translates textual ABS models into an internal representation and checks the models for syntax and semantic errors. The compiler back-end then generates code for the models targeting some suitable execution or simulation environment.

3.4.1 Syntax

Figure 3.15 specifies the ABS syntax related to delta modeling. Some of the nonterminals used herein refer to core ABS symbols, whose intended meaning should be immediate.

The *DeltaDecl* clause specifies the syntax of deltas, consisting of an unique identifier, a module access directive, a list of parameters and a sequence of module modifiers. The *ModuleAccess* directive gives the delta access to the namespace of a particular module. In other words, it specifies the ABS module to which the modifications specified by the delta apply by default. A delta can still apply changes to several modules by fully qualifying the *TypeName* of module modifiers.

The *ModuleModifier* clause describes the syntax of modifications at the level of modules. Such a modification can add a class or interface declaration, modify an existing class or interface, remove a class or interface, and also add functions, data types and type synonyms. Class modifications include the ability to change the interface of a class by adding or removing items from the class's list of implemented interfaces. The *InterfaceModifiers* clause describes how to modify existing interface declarations, either by adding new or removing existing method signatures.

The *Modifier* clause specifies the modifications that can occur within a class or interface body. These include adding and removing fields and methods, and

DeltaDecl ::= `delta` *TypeId* [*DeltaParams*] `;`
[*ModuleAccess*] *ModuleModifier**

ModuleModifier ::= `adds` *ClassDecl*
| `removes class` *TypeName* `;`
| `modifies class` *TypeName*
[`adds` *TypeId* (`,` *TypeId*)*]
[`removes` *TypeId* (`,` *TypeId*)*]
`{` *Modifier** `}`
| `adds` *InterfaceDecl*
| `removes interface` *TypeName* `;`
| `modifies interface` *TypeName* `{` *InterfaceModifier** `}`
| `adds` *FunctionDecl*
| `adds` *DataTypeDecl*
| `modifies` *DataTypeDecl*
| `adds` *TypeSynDecl*
| `modifies` *TypeSynDecl*
| `adds` *Import*
| `adds` *Export*

InterfaceModifier ::= `adds` *MethSig* `;`
| `removes` *MethSig* `;`

Modifier ::= `adds` *FieldDecl*
| `removes` *FieldDecl*
| `adds` *MethDecl*
| `removes` *MethSig*
| `modifies` *MethDecl*

DeltaParams ::= `(` *DeltaParam* (`,` *DeltaParams*)* `)`
DeltaParam ::= *Identifier HasCondition**
| *Type Identifier*

ModuleAccess ::= `uses` *TypeId* `;`

HasCondition ::= `hasField` *FieldDecl*
| `hasMethod` *MethSig*
| `hasInterface` *TypeId*

Figure 3.15: Delta grammar

modifying methods, which amounts to replacing a method implementation with a new one, while enabling the original method to be called using the **original** keyword. The aim of **original** is to enable the method being replaced to be called from the delta that replaces it. This is implemented by renaming the original method, and replacing the call via keyword **original** with a call to the renamed method. The semantics of calling **original** are essentially the same as `Super` from feature-oriented programming [11], and `proceed` from context-oriented programming [67], and similar to ordinary `super` calls in standard object-oriented languages, as well as `around` advice from aspect-oriented programming [74]. A *targeted* version of the **original** call is also provided. The target here is a specific delta or the core program (e.g. **core.original**`();` `Delta.`**original**`()`). This gives the user a tighter control over the program's behaviour. It also makes the code less flexible because calling **original** on a particular target delta introduces the assumption that the target delta has been already applied. Such a dependency could be invalidated for instance by changes to the SPL configuration, which dictates which deltas should be applied when certain features are selected.

In contrast to deltas presented in the literature [101, 102, 100], deltas in the ABS language can be parametrised by attribute values, which ultimately flow from the feature model's product selection. Delta parameters are discussed in Section 3.4.2.

Finally, the *HasCondition* describes constraints on class arguments to which a delta may be applied. These constraints consist of descriptions of the methods and fields such a class implements and any interfaces it is expected to have. For example:

```
delta D1 (C hasField Int f);
uses M; modifies class C { removes Int f; }

delta D2 (C hasMethod Unit setF(Int x));
uses M; modifies class C { modifies Unit setF(Int x) {...} }

delta D3 (C hasInterface T1);
uses M; modifies class C adds T2 removes T1 {...}
```

3.4.2 Delta Parameters

Deltas take an optional list of parameters. These are used to pass on configuration information defined in the product selection (cf. Section 3.1.2) down to the implementation level. Product declarations assign a boolean value

to each feature (true if selected, false otherwise), and a boolean or integer value to each feature attribute of features that are selected.

In the example below, any occurrence of the integer variable `param` (line 4) inside the delta module is replaced with the value of the feature attribute `F.x` (line 14) upon delta application. The concrete value of `F.x` depends on the selected product (lines 11–12). The connection between feature attributes and delta parameters is established in the SPL configuration (line 9), as described in Section 3.1.5.

```
1  module M;
2  // core classes
3
4  delta D(Int param);
5  adds class M.C { Int myField = param; }
6
7  productline PL;
8  features F;
9  delta D(F.x) when F;
10
11 product P1( F{x=0} );
12 product P2( F{x=17} );
13
14 root F { Int x in [0..99]; }
```

The boolean values of features can be accessed in similar fashion, as shown in the example below. Here, a delta adds three fields to class `C` (lines 6–8), which the configuration process assigns boolean constants based on the values of the delta parameters with the same name. These parameters reflect whether the features to which they are connected are selected or not (line 11). In this way, one can easily write code that reflects on the feature configuration.

```
module M;
class C {...}

delta D (Bool a, Bool b, Bool c);
modifies class M.C {
   adds Bool featureA = a;
   adds Bool featureB = b;
   adds Bool featureC = c;
}
productline PL; features A,B,C;
delta D(A,B,C) when A;

product P1(A);
product P2(A,B);
product P3(A,B,C);
```

3.4.3 Modifiers

Deltas can modify an ABS program in several ways. In general we distinguish between three types of modifications:

- an *addition* adds a new declaration (keyword **adds**),
- a *removal* removes an existing declaration (keyword **removes**),
- a *modification* changes an existing declaration (keyword **modifies**).

Corresponding to these modifications, deltas support three types of modifiers, which are declared using the respective keyword, as illustrated by the examples in this section.

Object-oriented Modifiers

To modify an object-oriented ABS program, deltas support adding new classes and removing existing ones. Existing classes can be also modified by adding new methods and also by removing or modifying existing methods. Deltas can also add new interface declarations, remove existing interface declarations, and modify interface declarations by adding or removing operations. Furthermore, deltas can change the interface of a class by adding or removing interfaces from the class's list of implemented interfaces. Lastly, deltas can introduce new fields to classes and remove existing fields.

Interfaces Deltas can introduce new interface declarations and remove or modify existing interface declarations. The syntax is illustrated by the following examples.

```
delta D1;
adds interface MyModule.I { Unit foo(); }

delta D2;
uses MyModule;
removes interface I;

delta D3;
uses MyModule;
modifies interface I { removes Unit foo(); adds Unit bar(); }
```

Classes Deltas can introduce new classes and remove existing classes. The syntax is illustrated by the following examples.

```
delta D1;
adds class MyModule.DataBase(Map<Filename,File> db) implements DB {...}

delta D2;
uses MyModule;
removes class Node();
```

The first delta `D1` above declares a new class `DataBase` inside the module `MyModule`. Delta `D2` removes the class `Node` from the same module. Specifying to which module such code modifications apply can be done in two ways. First, as exemplified by delta `D1`, the class name can be qualified with a module name. An alternative way is to include a **uses** `<moduleName>` clause at the beginning of the delta declaration. This instructs the delta to open the specified module in order to perform changes. In this case modifiers don't need to qualify the names of interface or classes they refer to. When a delta specifies modifications to a single module, this method is more concise. When a delta specifies modifications across multiple modules, it is more convenient to qualify each class modifier with a module name. Using both methods together is also possible, in which case unqualified class names will refer to classes defined inside the *used* module.

Deltas can also *modify* existing classes by adding new methods and removing or modifying methods; by adding or removing fields; and by manipulating the list of interfaces that the class implements. These operations are illustrated in the following sections.

Methods Methods can be added, removed or modified from within deltas. The following example shows a delta designed to modify the behaviour of the class `Greeter` by modifying its `sayHello` method. The class is assumed to have been declared in the core program inside the `Hello` module.

```
delta Nl;
uses Hello;
modifies class Greeter {
   modifies String sayHello() {
      return "Hallo wereld";
   }
}
```

The above `Nl` delta applies its changes to the core ABS module `Hello`, as specified by the **uses** clause. It provides a new implementation for the method `sayHello()` in class `Greeter` by declaring a so-called method *modifier*. The method modifier is introduced by the **modifies** keyword and followed by the method signature and a block of code providing the method's new implementation.

Adding entirely new methods is also supported using the **adds** keyword followed by the method signature and its implementation. Similarly, it is possible to remove methods from classes using **removes** followed by the method signature, as shown in the following.

```
delta D;
modifies class M.Foo {
   adds Int bar() { return 17; }
   removes Unit moo();
}
```

Calling original By calling **original** from within a method modifier body, it is possible to access the method's previous behaviour, that is, the behaviour implemented in the previously applied delta or in the core. This is similar to calling `super` to access the superclass behaviour of a method in a language with class inheritance such as Java. An **original** call has to supply a list of arguments that conforms with the original method's list of formal parameters.

Targeted original calls Original calls can be targeted towards a given delta by prefixing the call with the name of the delta, or towards the core ABS code by using the keyword **core**:

```
core.original(params);
Delta.original(params);
```

Regular (untargeted) **original** calls invoke the method behaviour defined by the previously applied delta. For example, if a method `m` is defined in the core, and then a set of deltas `D1..D3`, which each modify `m`, are applied in sequence, then calling **original** from within `m`'s modifier in `D3` will run the version of `m` defined in `D2`. With a targeted call, one can access any version of `m`, that is, the versions defined in `D2`, `D1` and in the core.

This allows a tighter control of which code is actually executed when calling **original**. As the order of delta application is often not uniquely defined (the SPL configuration defines only a partial order), it is not always determinable which behaviour will be invoked upon calling **original**. With a targeted original call, the user can specify exactly which code to execute and even invoke multiple versions of a method. This, of course, requires that the target delta has been applied already; otherwise the compiler will indicate an error.

```
module M;
class C {
   String m(String s) { return(s) };
}
delta D1;
modifies M.C {
   modifies String m(String s) { return prefix + original(s); }
}
delta D2;
modifies M.C {
   modifies String m(String s) { return original(s) + suffix; }
}
delta Resolve;
modifies M.C {
   modifies String m(String s) { return prefix + core.original(s) + suffix; }
}
```

Consider the above example. `D1` and `D2` both modify method `m` in different, non-compatible ways. We say that these two deltas are in conflict. Assume that `D1` and `D2` can be applied in any order, and that delta `Resolve` has to be applied after `D1` and `D2`. By calling **original** from within `Resolve`, we cannot be sure which version of `m` will actually be invoked: this depends on whether `D1` or `D2` has been applied last. By targeting the original call towards a specific delta, we can control the behaviour precisely, and resolve the conflict in a meaningful way.

Targeted original calls were required for the implementation of the delta modelling workflow (DMW) [61, 63]. The DMW describes a process of applying delta modelling to obtain a model of a software product line that is globally unambiguous and complete. A focus of DMW is the systematic reconciliation of conflicting feature functionality.

Class interfaces A delta can change the list of interfaces that a class implements. Adding or dropping interfaces from that list is achieved using the familiar **removes** and **adds** keywords.

The following example shows a core ABS program defining a `Logger` class that implements the `Output` interface. It further declares a delta that modifies the `Logger` class so that it implements a different interface. This new `IO` interface is introduced in the same delta.

```
module M;
interface Input { String read(); }
interface Output { Unit write(String s); }
class Logger implements Output {
   Unit write(String s) {...}
}

delta IO;
adds interface IO extends Input, Output {}
modifies class Logger adds IO removes Output {
   adds String read() {...}
}
```

Fields In addition to modifying object behaviour, ABS allows adding or removing fields. New fields are introduced by the **adds** keyword followed by the field's type, name, and an optional value assignment. Similarly, fields can be removed using the **removes** keyword. The following example demonstrates this.

```
delta D;
modifies class M.Foo {
   adds List<Item> items;
   adds Int itemsCount = items.size();
   removes String name;
}
```

Functional Modifiers

Functional program elements can also be modified from within deltas. ABS supports the addition of functions, algebraic data types and type synonyms, and the modification of algebraic data types and type synonyms. Qualifying functional elements with the module name is currently unsupported, therefore when adding functional elements, a **uses** clause has to be specified.

Functions Example of adding a function.

```
delta MyDelta;
uses MyModule;
adds def Int min(Int a, Int b) = case a < b { True => a; False => b; };
```

Data types Example of adding a data type.

```
adds data Schedule = Schedule(
   String schedname,
   List<Item> items,
   Int sched,
   Deadline dline) | NoSchedule;
```

The `Schedule` data type added above could be later modified as shown in the example below.

```
modifies data Schedule = Schedule(
   String schedname,
   List<Item> items,
   Int sched,
   Deadline dline) | NoSchedule(String reason);
```

When modifying a datatype, the given constructors supersede the previous list of constructors.

Type synonyms Example of adding a type synonym.

```
adds type ClientId = Int;
```

Example of modifying a type synonym.

```
modifies type ClientId = String;
```

Module Modifiers

ABS offers a module system for creating namespaces, and structuring and hiding code. Deltas support, to some extent, modifications to the modules defined by an ABS model.

Imports and Exports The addition of (qualified and unqualified) **import** and **export** statements to ABS modules is supported, as shown in the following examples.

```
delta D1;
uses Drinks;
adds export Drink, Milk;

delta D2;
uses Bar;
adds export *;
adds import Drinks.Milk;

delta D3;
uses MyModule;
adds import * from Bar;
```

The **adds import** and **adds export** directives apply to the module defined by the **uses** statement.

Unsupported Modifications

While delta modelling supports a broad range of ways to modify an ABS model, not all ABS program entities are modifiable. These unsupported modifications are listed here for completeness. While these modifications could be easily specified and implemented, we opted not to overload the language with features that have not been regarded as necessary in practice.

Class parameters and init block ABS class parameters define additional fields of the class; an initialisation block has a purpose similar to that of a constructor in other languages. Deltas currently do not support the modification of class parameter lists or class init blocks.

Functional program elements Deltas currently only support adding functions, and adding and modifying data types and type synonyms. Removal is not supported.

Modules For name spacing, code structuring, and code hiding purposes, ABS offers a module system that supports name importing and exporting [1]. Deltas currently do not support adding new modules or removing modules.

Imports and Exports While deltas do support the addition of **import** and **export** statements to modules, they do not support their modification or removal.

Main block An ABS main blocks is similar to Java's `Main` class. Deltas currently do not support the modification of the program's main block.

3.4.4 Semantics

This section presents a formal semantics of deltas. Applying a delta Δ to a core ABS program P yields a new core ABS program. Thus a product is constructed by successively applying deltas, one at a time, to a core program.

The formalisation is based on the more abstract presentation of Clarke et al. [28]. That work also describes the composition of deltas with each other, which is essential for reasoning about conflicting delta modules, but this feature is elided from the current presentation.

ABS programs, classes and deltas will be represented in terms of finite maps from identifiers to the corresponding contents of the program, class, or delta, in order to more cleanly present the semantics. The semantics only describes the modifications of methods; dealing with fields, functions, and so forth is a straightforward extension. Parameters are omitted.

Let *CIdentifier*, *MIdentifier*, and *DIdentifier* be the set of identifiers for classes, methods, and deltas, respectively, and let *MethBody* be the set of method bodies, including the parameter and return types, with possible references to (targeted and untargeted) **original** methods. In the following domains, Modify

and Remove are used to tag the various branches of sum data types.

$$\begin{aligned}
\mathit{Program} &= \mathit{CIdentifier} \rightharpoonup \mathit{ClassBody} \\
\mathit{ClassBody} &= \mathit{MIdentifier} \rightharpoonup (\mathit{MethBody} \times \mathit{DIdentifier}) \\
\mathit{Delta} &= \mathit{CIdentifier} \rightharpoonup \mathit{DeltaBody} \\
\mathit{DeltaBody} &= \mathsf{Modify}\ (\mathit{MIdentifier} \rightharpoonup ((\mathit{MethBody} \uplus \mathsf{Remove}) \times \mathit{DIdentifier})) \\
&\quad \uplus\ \mathsf{Remove}
\end{aligned}$$

A program is a map from class names to classes. Class bodies are collections of pairs of a method body and the identifier of the delta used in its creation or last update. Initially all class bodies have their methods associated to the special delta identifier **core**. A delta is a map from class names being modified to delta bodies. Note that, for technical convenience, the *DIdentifier* is included in the delta bodies and not in *Delta*. Delta bodies consist of two different types of modification: Modify modifies a class in place or creates a new class if it does not exist, where the two elements within a Modify clause correspond to either (1) replacing or adding a method with a new body from *MethBody*, or (2) removing the method. Finally, Remove denotes the removal of the class.

Notation 3.4.1. *Let $f : X \rightharpoonup Y$ denote a partial function from X to Y. If $f(x)$ is undefined for $x \in X$, write $f(x) = \bot$, where $\bot \notin Y$. For set A, let $A_\bot$ denote $A \cup \{\bot\}$, where $\bot \notin A$. We freely shift between partial functions $X \rightharpoonup Y$ and functions $X \to Y_\bot$. If $\odot : A_\bot \times B_\bot \to C_\bot$, define the lifting of $\odot$ to partial functions over index set I as*

$$- \mathbin{\overline{\odot}} - : (I \rightharpoonup A) \times (I \rightharpoonup B) \to (I \rightharpoonup C)$$

$$(f \mathbin{\overline{\odot}} g)(i) = f(i) \odot g(i), \qquad \text{where } i \in I.$$

Given class update $u : \mathit{MIdentifier} \rightharpoonup ((\mathit{MethBody} \uplus \mathit{Remove}) \times \mathit{DIdentifier})$, define function $u^* : \mathit{MIdentifier} \rightharpoonup (\mathit{MethBody} \times \mathit{DIdentifier})$ as follows. For $i \in \mathit{MIdentifier}$:

$$u^*(i) = \begin{cases} \bot & \text{if } u(i) = (m, d) \text{ and } m \in \mathit{Remove}, \\ u(i) & \text{otherwise.} \end{cases}$$

Notation 3.4.2. *Given $d \in \mathit{DIdentifier}$ and $i \in \mathit{MIdentifier}$, let $\epsilon(i, d) \in \mathit{MIdentifier}$ be a method identifier uniquely defined by d and i. Given class update $u : \mathit{MIdentifier} \rightharpoonup ((\mathit{MethodBody} \uplus \mathit{Remove}) \times \mathit{DIdentifier})$ and class body $c : \mathit{MIdentifier} \rightharpoonup (\mathit{MethBody} \times \mathit{DIdentifier})$, define function $\xi(u, c) : \mathit{MIdentifier} \rightharpoonup (\mathit{MethBody} \times \mathit{DIdentifier})$ as follows.*

$$\xi(u, c) \;=\; \{\epsilon(i, d) \mapsto (m, d) \mid (i \mapsto (m, d))) \in c, i \in \mathit{dom}(u)\}$$

Notation 3.4.3. *Given $m \in \mathit{MethodBody}$, $i \in \mathit{MIdentifier}$, and $d \in \mathit{DIdentifier}$, $m[i,d] : \mathit{MethodBody}$ denotes the method body m after replacing all occurrences of* **original** *by $\epsilon(i,d)$ and all occurrences of* `target.`**original** *by $\epsilon(i,$`target`$)$.*

Definition 3.4.4 (Delta module application)**.** *The application of a delta to a program is specified by the following functions:*

$$\begin{aligned} \mathit{apply} &: \mathit{Delta} \times \mathit{Program} \rightharpoonup \mathit{Program} \\ \mathit{apply}(d,p) &= d \mathbin{\overline{\odot_c}} p \end{aligned}$$

where $- \odot_c - : \mathit{DeltaBody}_\perp \times \mathit{ClassBody}_\perp \to \mathit{ClassBody}_\perp$

$$\begin{array}{rcl@{\qquad}rcl} \perp \odot_c\ x & = & x & (\mathit{Modify}\ u) \odot_c \perp & = & u^* \\ \mathit{Remove} \odot_c\ \_ & = & \perp & (\mathit{Modify}\ u) \odot_c\ c & = & u \mathbin{\overline{\odot_m}} c \cup \xi(u,c) \end{array}$$

and $- \odot_m - : ((\mathit{MethBody} \uplus \mathit{Remove}) \times \mathit{DIdentifier})_\perp \times (\mathit{MethBody} \times \mathit{DIdentifier})_\perp \to (\mathit{MethBody} \times \mathit{DIdentifier})_\perp$

$$\begin{array}{rcl@{\qquad}rcl} \perp \odot_m\ x & = & x & (m,d) \odot_m \perp & = & (m,d) \\ (\mathit{Remove}, d) \odot_m\ \_ & = & \perp & (m,d) \odot_m\ (m',d') & = & (m[m_{id}, d'], d) \end{array}$$

where $m \in \mathit{MethBody}$, $m_{id} \in \mathit{MIdentifier}$ is the identifier of method m, and $d, d' \in \mathit{DIdentifier}$.

Implementation In practice, for every delta body *Modify* u for class C and for each $m_{id} \to (m,d)$ with $m \in \mathit{MethBody}$, the following steps are performed:

1. if exists $m' \in \mathit{MethBody}$ and $d' \in \mathit{DIdentifier}$ such that $m_{id} \mapsto (m',d')$ is in the class body of C, then:
   - replace it with $m_{id} \mapsto (m[m_{id}, d'], d)$, and
   - add $\epsilon(m_{id}, d') \mapsto (m', d')$;
2. otherwise add $m_{id} \mapsto (m,d)$.

The targeted original calls example in Section 3.4.3 (page 62) illustrates this process with concrete code. The modified method in that example is `m`, belonging to class `C`. Originally `C` $= \{m_{id} \mapsto (m,$ `core`$)\}$, where m is the method body of `m` and m_{id} is its identifier. A possible sequence of applying the three deltas is (`D1`, `D2`, `Resolve`).

Application of `D1`. Let m_1 be the new method body of `m` defined in `D1`. First we calculate $m_1[m_{id}, \texttt{core}]$ by replacing **original** by $\epsilon(m_{id}, \texttt{core})$ in m_1, using $\epsilon(m_{id}, \texttt{core}) = \texttt{m\$ORIGIN\_core}$. Second we add $m_{id} \mapsto (m_1[m_{id}, \texttt{core}], \texttt{D1})$ and $\epsilon(m_{id}, \texttt{core}) \mapsto (m, \texttt{core})$ to class `C`.

Application of `D2`. Let m be the method body for `m` after applying `D1` and m_2 be the new method body of `m` defined in `D2`. First we calculate $m_2[m_{id}, \texttt{D1}]$ by replacing **original** with $\epsilon(m_{id}, \texttt{D1})$ in m_2, using $\epsilon(m_{id}, \texttt{D1}) = \texttt{m\$ORIGIN\_D1}$. Second we add $m_{id} \mapsto (m_2[m_{id}, \texttt{D1}], \texttt{D2})$ and $\epsilon(m_{id}, \texttt{D1}) \mapsto (m, \texttt{D1})$ to class `C`.

Application of `Resolve`. Let m be the method body for `m` after applying `D2` and m_r be the new method body of `m` defined in `Resolve`. First we calculate $m_r[m_{id}, \texttt{D2}]$ by replacing **core.original** with $\epsilon(m_{id}, \texttt{core})$ in m_r. Second we add $m_{id} \mapsto (m_r[m_{id}, \texttt{D2}], \texttt{Resolve})$ and $\epsilon(m_{id}, \texttt{D2}) \mapsto (m, \texttt{D2})$ to class `C`.

The application of the three deltas results in the following code.

```
module M;
class C {
   String m(String s) { return prefix + m$ORIGIN_core(s) + suffix; };
   String m$ORIGIN_core(String s) { return(s); }
   String m$ORIGIN_D1(String s) { return prefix + m$ORIGIN_core(s); }
   String m$ORIGIN_D2(String s) { return m$ORIGIN_D1(s) + suffix; }
}
```

After the application of all three deltas, the class `C` has four methods: `m$ORIGIN_core`, `m$ORIGIN_D1`, `m$ORIGIN_D2` and `m`. Both `m` and `m$ORIGIN_D1` call `m$ORIGIN_core`, while `m$ORIGIN_D2` calls `m$ORIGIN_D1`. Observe that the method `m$ORIGIN_D2` is added to `C` but never called. A simple optimisation is to postpone its addition to `C` until it is called from within any method body. Furthermore, unreachable versions of methods can be safely removed.

3.5 SPL Configuration

This section describes the product line configuration language, which links feature models specified in μTVL (Section 3.2) with deltas (Section 3.4) to specify the variability in a product line. This approach is similar to the product line specification proposed in more recent versions of delta-oriented programming [102, 100], but we add a more explicit syntax for ordering the application of deltas.

Configuration	::=	`productline` *TypeId* `;` *Features* `;` *DeltaClause**
Features	::=	`features` FID (`,` FID)*
DeltaClause	::=	`delta` *DeltaSpec* [*AfterClause*] [*WhenClause*] `;`
DeltaSpec	::=	*TypeName* [`(` *DeltaParams* `)`]
DeltaParams	::=	*DeltaParam* (`,` *DeltaParam*)*
DeltaParam	::=	FID | FID.AID
AfterClause	::=	`after` *TypeName* (`,` *TypeName*)*
WhenClause	::=	`when` *ApplicationCondition*
ApplicationCondition	::=	*ApplicationCondition* `&&` *ApplicationCondition*
	|	*ApplicationCondition* `||` *ApplicationCondition*
	|	$\sim$ *ApplicationCondition*
	|	`(` *ApplicationCondition* `)`
	|	FID

Figure 3.16: SPL configuration grammar

A product line configuration consists of a set of features assumed to exist and a set of *delta clauses*. Each delta clause specifies a delta and the conditions required for its application, propositional formulas over the set of known features and attributes called *application conditions*, and a partial ordering relation with respect to other deltas. When the propositional formula holds for a given product, the delta is said to be active. The partial order states which deltas, when active, should be applied before the current delta.

3.5.1 Syntax

The syntax of the product line configuration language is given in Figure 3.16. The *Configuration* clause specifies the name of the product line, the set of features it implements, and the set of deltas used to implement those features. The *Features* clause describes which features the product line refers to. These are included so that certain simple self-consistency checks can be performed. The *DeltaClause* is used to specify each delta, linking it to the feature model. Each *DeltaClause* has a *DeltaSpec*, specifying its name and its parameters, an *AfterCondition*, specifying the deltas that the current delta must be applied after, and an *ApplicationCondition*, specifying an arbitrary predicate over the feature and attribute names (see Figure 3.9) that describes when the given delta is included in the product line.

Figure 3.17 shows how the Hello World product line is configured, connecting

```
productline MultiLingualHelloWorld;
features English, German, Dutch, Repeat;
delta Rpt(Repeat.times) after De, Nl when Repeat;
delta De when German;
delta Nl when Dutch;
```

Figure 3.17: Configuration of "Hello World" SPL

the features and attributes defined in the feature model to deltas. It first names the set of features (from the feature model in Figure 3.10) used to configure this product line. The **delta** clauses link each delta to the feature model through an application condition (**when** clause); in this case, a delta module is applied simply when the specified feature is selected (e.g. `De` **when** `German`). There is no delta corresponding to the feature `English`, as the core module provides support for the English language by default. In addition, `Rpt` has to be applied **after** `De` and `Nl`. `Rpt`'s argument is `Repeat.times`, the `times` attribute feature `Repeat`; its value (defined by product selection, see Section 3.3) is propagated to the `Rpt` delta.

3.5.2 Semantics

A SPL configuration specifies how the feature model relates to the deltas that are to be applied to the core module. It does so by specifying the parameters and application conditions for each delta, and by ordering the deltas.

Each delta referred to in a configuration file is modelled by an element of the following type:

$$\mathit{Delta} \times \mathit{Params} \times \mathit{AppCondition}$$

where *Delta* is the semantic domain of delta bodies, defined in Section 3.4.4,

$$\mathit{Params} \;=\; \mathit{Var} \rightharpoonup \mathsf{FID} \uplus (\mathsf{FID} \times \mathsf{AID}) \uplus \mathsf{Int}$$

models the substitution of actual parameters, which may be attributes or constants, defined in the configuration script with the formal parameters of the corresponding delta, and *AppCondition* is the syntactic category of application conditions.

An SPL configuration can be modelled as a partial order over the declared deltas (with their parameters and application conditions), where the partial order is determined by the reflexive, transitive closure of the **after** clauses. This

is given by the following domain, where $PO(-)$ denotes the collection of all partial orders over a given set:

$$\mathit{Config} = \mathit{PO}(\mathit{Delta} \times \mathit{Params} \times \mathit{AppCondition}).$$

The semantics of a configuration script $\mathit{conf} \in \mathit{Config}$ is a function of type

$$[\![\mathit{conf}]\!]\_ : \mathit{ProductSelection} \to \mathcal{P}(\mathit{Delta}^*)$$

which maps the interpretation of a product selection (see Section 3.3.2) to the deltas to apply, in the order they should be applied. Note that many orders may exist if the **after**-order is underspecified. A product selection is an assignment from feature names to true or false (1 or 0) and from attributes to values, given by the domain *ProductSelection*:

$$\mathit{ProductSelection} = (\mathsf{FID} \uplus (\mathsf{FID} \times \mathsf{AID})) \rightharpoonup \mathsf{Int}$$

We now develop the ingredients making up function $[\![\mathit{conf}]\!]\_$.

First, assume that a notion of substitution exists for deltas, respecting the scoping of variables, to replace parameters with appropriate values:

$$\mathit{Subst} = \mathit{Var} \rightharpoonup \mathsf{Int}$$

$$\mathit{applySubst} \ : \ \mathit{Subst} \times \mathit{Delta} \to \mathit{Delta}$$

Next, we define the composition of the parameter specifications of deltas with a product selection, giving a mapping from formal parameters of delta modules to values (Int), which will be used to refine the deltas with the configuration parameters specifying in the product selection:

$$\circ \ : \ \mathit{ProductSelection} \times \mathit{Params} \to \mathit{Subst}$$

$$\sigma \circ p = \{v \mapsto x\sigma \ \mid \ v \mapsto x \in p\}$$

$$\text{where } x\sigma = \begin{cases} v & \mathit{if}\ x \in \mathsf{FID} \uplus (\mathsf{FID} \times \mathsf{AID}))\ \mathit{and}\ x \mapsto v \in \sigma \\ x & \mathit{if}\ x \in \mathsf{Int} \end{cases}$$

Now the function taking a product selection $\sigma \in \mathit{ProductSelection}$ and giving the collection of deltas to apply is computed as the composition of the following steps:

1. Select applicable deltas by applying $select\_ : \mathsf{Config} \to \mathsf{PO}(\mathsf{Delta} \times \mathsf{Params})$

   $select_\sigma(D, \prec) = (D', \prec|_{D'})$,

   where $D' = \{(d, p) \mid (d, p, \phi) \in D, \sigma \models \phi\}$ and $\prec|_{D'}$ is $\prec$ restricted to D',

   and $\models \subseteq \mathsf{ProductSelection} \times \mathsf{AppCondition}$ is the satisfaction relation.

2. Specialise deltas using the function $specialise : \mathsf{ProductSelection} \times \mathsf{PO}(\mathsf{Delta} \times \mathsf{Params}) \to \mathsf{PO}(\mathsf{Delta})$

   $specialise_\sigma(D, \prec) = (D', \prec|_{D'})$, where $D' = \{applySubst(\sigma \circ p, d) \mid (d, P) \in D\}$.

3. Order deltas using the function $order : \mathsf{PO}(\mathsf{Delta}) \to \mathcal{P}(\mathsf{Delta}^*)$

   $order((D, \prec)) = \{[d_1, \ldots, d_n] \mid d_1, \ldots, d_n$ is a linear extension of $(D, \prec)\}$.

Finally, the semantics of an SPL configuration can be interpreted as a function

$$[\![\_]\!]\_ : \mathsf{Config} \times \mathsf{ProductSelection} \to \mathcal{P}(\mathsf{Delta}^*)$$

$$[\![conf]\!]_\sigma = order(specialise_\sigma(select_\sigma(conf))).$$

Note that this process may be ambiguous when multiple orderings of deltas are possible. This should be resolved either by adding more elements to the **after** order or by introducing conflict-resolving deltas [28].

3.6 Tool Support

The ABS modelling language is accompanied by a tool framework [122] consisting of several tools (implemented in Java) to produce, analyse, and use ABS models. Such tools include a compiler front-end, code generators, an Eclipse IDE, a unit testing framework, a package dependency manager, a debugger and visualiser and a static resource analysis tool. A foreign function interface makes ABS code interoperable with Java. We focus on the compiler and particularly on its ability to *flatten* an ABS model as part of configuring a software product for deployment.

3.6.1 The ABS Compiler Framework

The compilation process is summarised in Figure 3.18. The ABS parser generates an "extended" AST (abstract syntax tree), which includes the variability

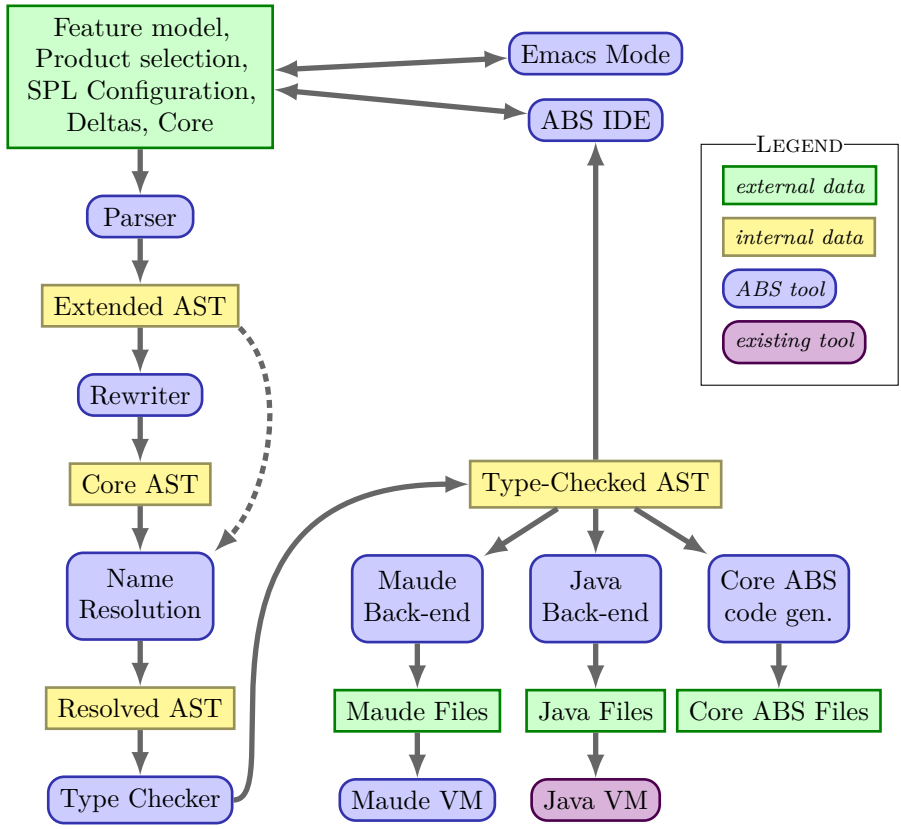


Figure 3.18: Overview of the ABS compiler framework

information defined by the feature model, product selection, deltas and SPL configuration.

The compiler front end deals with the variability encoded in ABS models in one of two ways. In the first case, the variability-enhanced AST is flattened by applying the deltas that correspond to the configured feature set to the core, thereby obtaining a "core AST". After name resolution and type checking, the compiler back-end then generates the executable software product using one of several code generation engines such as the Maude or Java code generators. This process describes *static* product generation, where product configuration is part of the compilation process. Characteristically for static product configuration, all variability information contained in the model is removed at compile-time. The result is a system that represents a single particular software product of the underlying SPL.

The ABS tool framework supports a second, alternative approach, which preserves the variability information with the purpose of exploiting it at runtime. This approach is illustrated in Figure 3.18 by the dotted arrow, which bypasses the rewriting of the extended AST into a core AST. Hence code is generated based on the extended AST, which includes the variability model. This approach effectively enables the generation of dynamic software product lines (DSPLs). DSPLs and runtime variability are the subject of Chapter 4.

3.6.2 Using the ABS Compiler

The compiler front end is, in general, used as part of the ABS IDE, but it can also be used as a stand-alone tool on the command line.

Analysing Feature Models

Product selections are validated with respect to a given feature model using the ABS tools. For example, the following command verifies that the product `HighEnd` belongs to the Chat SPL feature model:

```
java -cp absfrontend.jar abs.frontend.parser.Main \
    -check=HighEnd ChatPL.abs
```

Besides simply validating products, the ABS compiler can perform other analysis tasks on a given feature model, such as solving the constraint satisfaction problem (CSP), obtaining all solutions for the CSP, obtaining a solution that includes a given product, minimising and maximising certain variables, obtaining the solution(s) with the most features, or finding a solution that stays as close as possible to a given (possibly invalid) product.

Flattening ABS models

Flattening an ABS model means applying a sequence of deltas to a core ABS model, in order to obtain the behaviour of a particular product. In the ABS compiler front end, the `-product=<name>` switch triggers the flattening for a given product, as shown in the example below.

```
java -jar absfrontend.jar -product=P1 HelloWorld.abs
```

If the application of deltas, name resolution or type checking are not successful, an appropriate message is displayed. Otherwise, no output is displayed and

the internal AST is flattened according to the product selection. The flattened AST can then be used, for example, in the generation of Java code.

3.7 Discussion

This section discusses design aspects of the ABS variability modelling framework, the advantages and limitations that result from these, and points to studies that apply ABS's variability modelling capabilities in practice.

3.7.1 Granularity of Delta Transformations

When designing a program transformation methodology such as delta modelling, the granularity of supported transformations has to be decided. This granularity represents a trade-off between the complexity of the transformation language and its potential to minimise code duplication and facilitate code reuse. Deltas enable the modification of object-oriented ABS programs at the level of fields and methods, and for functional programs at the level of functions and data types. For example, if a delta needs to modify a single statement in the body of a method, generally, it needs to provide the entire code of that method's body, including the statements that remain unchanged. This circumstance is mitigated by the possibility to access the method's old behaviour by calling **original**. This can be used with method modifications that wrap additional code around the code of the original method. The granularity of deltas is in line with other program transformation/synthesis techniques such as class inheritance, traits [18], mixins [22] and feature refinements [11]. Some implementations of aspects [74] allow a finer granularity. It has been argued that method-level granularity is too coarse when the task is to re-engineer a legacy application into an SPL [73]. A finer-grained association of features with code can be typically achieved by using code annotations (Section 3.8.1) rather than a compositional approach such as ours.

3.7.2 Quality Assurance

The variability inherent to SPL adds complexity to the engineering process and to quality assurance tasks in particular. The family engineering process requires dedicated analysis methods that are able to guarantee certain safety properties (e.g. feature model validity, type safety) for the entire product line in an efficient manner.

A feature model specified in μTVL is represented internally as a constraint over a collection of boolean and integer variables (i.e. the features and attributes). This makes it straightforward to encode certain analysis tasks as constraint satisfaction problems and solve these using CSP methods. The ABS tool suite implements a set of basic feature model analysis tasks, including the validation of the products specified in a product selection, identifying all products supported by the model and finding all products when a partial feature selection is given. Many other analysis methods exist [14] and are being considered for inclusion in the ABS framework.

Software products generated from an SPL developed in ABS are type checked using the static type system of the ABS language *after* the flattening process. This limits the potential of ABS to implement large SPL models: generating and type checking all variants quickly becomes impractical. In this light, the challenge when type checking an entire SPL is to ensure that all its products are type safe without having to generate each product. Developing such type systems has only recently shifted into focus. Some aspects of compositional type checking have been studied in the context of delta oriented programming [79, 100, 19], but a comprehensive solution is still in the future.

Another class of quality properties is related to ensuring consistency between the feature model and the delta code base. ABS implements a number of checks based on the SPL configuration, such as the applicability of deltas in the context of the given feature model. By implementing further analysis measures, such as checking whether each product has a unique implementation, will strengthen the connection between problem and solution domain models in ABS.

3.7.3 Evaluation

The variability modelling extension of the ABS language has been evaluated in a series of case studies by the HATS project's industrial partners [6, 123, 122].

In the context of modelling an industrial application containing 21 features and 768 products, the μTVL feature modelling and product selection languages were found to provide the necessary expressiveness; the available tool support was deemed "very usable" [6]. ABS's holistic approach to expressing variability using features and relating them to object behavior using deltas and the SPL configuration was highlighted. With respect to behavioural modelling, the evaluation considered expressiveness, configuration, modularity, reusability and tool support and came to a positive assessment [6]. The feedback we received during these evaluations was used to fine-tune and improve the language and tools.

The ABS delta modelling implementation is also the basis for evaluating the delta modelling paradigm in practical terms. While delta modelling offers expressivity and great flexibility in designing and implementing variable systems, it is only through practical application that best practices and recommended patterns of good delta design can be established. The Delta Modelling Workflow (DMW) [61, 63] is a guided process that allows a developer to create a product line from scratch that is globally unambiguous and complete. Global unambiguity guarantees that each feature configuration generates a unique product and completeness ensures that each product satisfies the specifications of the features that it implements. This is achieved by systematically detecting any conflicts between deltas and resolving them using "conflict resolving" deltas.

ABS deltas enabled the development of a white-box unit testing framework for ABS called ABSUnit [4]. Implementing tests often requires to access or to change class internals (e.g., to check intermediate results or to shortcut complex initialization procedures). Deltas provide an elegant solution: instead of cluttering the code base with auxiliary code, all test-related changes are organized into separate deltas. Those deltas are only selected during product testing, but are absent from the actually shipped product. In short, in ABS test code becomes a product feature.

3.8 Related Work

Existing approaches to express variability in modelling languages can be classified in two main directions [117]: annotative (or negative) and compositional (or positive). A third main approach for representing variability of development artifacts are model transformations, in which variability is represented by transforming a base model to obtain a product variant. Delta modelling is a transformational approach [103].

3.8.1 Annotations

Annotative approaches consider one model representing all products of the product line. Variant annotations, for example, using UML stereotypes in UML models [125, 52] or presence conditions [38], define which parts of the model have to be removed to derive a concrete product model. The orthogonal variability model (OVM) proposed by Pohl. et al. [94] models the variability of product line artifacts in a separate model where links to the artifact model take the place of annotations. Similarly, decision maps in KobrA [9] define which parts of the product artifacts have to be modified for certain products. In

the Koala component model [115], the variability of a component architecture containing all possible components is expressed by component parametrisation that is instantiated depending on the product features.

3.8.2 Composition

Compositional approaches associate model fragments with product features that are composed for a particular feature configuration. A prominent example of this approach is AHEAD [11], which can be applied on the design as well as on the implementation level. In AHEAD, a product is built by stepwise refinement of a base module with a sequence of feature modules. Design-level models can also be constructed using aspect-oriented composition techniques [60, 117, 90]. Apel et al. [5] apply model superposition to compose model fragments.

3.8.3 Transformations

The common variability language (CVL) [58] represents the variability of a base model by rules describing how modelling elements of the base model have to be substituted in order to obtain a particular product model. Jayaraman et al. [68] define graph transformation rules that capture artifact variability of a single kernel model comprising the commonalities of all systems. Hendrickson et al. [64] represent architectural variability by change sets containing additions, removals or modifications of components and component connections that are applied to a base line architecture. Perrouin et al. [93] obtain a product model by model composition and subsequently refinement by model transformation.

Delta Modelling

The notion of program deltas was introduced by Lopez-Herrejon [81] to describe the modifications of object-oriented programs. Schaefer et al. [104, 99] introduced delta modelling as a means to develop product line artifacts suitable for automated product derivation. The conceptual ideas of delta modelling have also been applied the programming language level in an extension of Java with core and delta modules allowing the automatic generation of Java-based product implementations [101]. In later work, Schaefer et al. [102, 100] propose a version of delta-oriented programming where products are generated only from delta modules applied to the empty product. Furthermore, in this version the application conditions and the application ordering are specified separately from the delta modules in a product line specification in order to

increase the reusability of the delta modules and to enable compositional type checking. Ensuring that an SPL is type safe without having to build and check each product individually is an important problem. A foundational calculus has been proposed for ensuring type safety of delta-oriented SPL developed in Java [100, 19]. In a further line of work, the delta-oriented programming paradigm has been extended to support dynamic reconfiguration by adding a dynamic reconfiguration graph that specifies how to switch between different feature configurations [40, 39].

3.9 Summary

This chapter presents the variability modelling capabilities of the ABS language that add dedicated support for the development of software product lines. ABS supports the modelling of SPLs both in the problem and in the solution domain, and establishes a connection between these two.

The language constructs pertaining to the problem space (i.e., user requirements) are the feature modelling language μTVL and a product selection facility. On the solution (i.e., design and implementation) side, deltas are used to encapsulate variable code components. An SPL configuration language connects feature models and deltas, ensuring that the implemented solution can be traced to the problem space variability model (and vice versa). Due to this connection, when refining or updating the problem domain model, it will be immediately evident that the solution has to be adapted accordingly.

Traceability between problem and solution domains helps accomplish a second important goal in SPL engineering: when the user selects a particular product based on a set of desired features, the appropriate software product can be automatically generated by the ABS compiler.

Chapter 4

Language Design for Dynamically Adaptable Software

Variability in software is most often motivated by diversified application contexts or user requirements. This is sometimes referred to as spatial variability [29] and is addressed by the ABS variability framework presented in the previous chapter. However, it is also common that the application context or requirements of one and the same user vary *over time*. This kind of variability is known as *temporal* variability or evolvability [27]. Systems that support temporal variability need to provide a more flexible configuration mechanism that allows them to adapt to required changes in a timely manner.

The flexibility of exploiting a system's variability depends on the points in time during development, deployment, and use at which the variable parts of a system are selected, the so-called *binding time*. Arguably the most widely supported binding time in variability-enabled systems is development time, or, more specifically, compile time. Compile-time variability is resolved by selecting appropriate variants of the system's source code components and then compiling these together to produce the executable product. While resolving variability at compile time is often desirable, as it removes complexity from the system, it can also be too restricting. Compiling software generally precedes its distribution to customers, therefore, a customer will not be able to exploit the system's variability, as it will have been already removed before the system was delivered. Hence, when a customer's requirements change, he has to obtain a different

variant, configured to address his new requirements. This incurs, at the very least, some cost and downtime related to replacing an old system with a new one.

Binding variable software components together can also happen at any stage *after* compilation. Available literature distinguishes between several binding times ranging from design time, compile time, via linking time, startup time, initialisation time, to runtime [106]. In this chapter we focus on variability *at runtime*, from the perspective of designing a language that gives the developer control over different aspects of program variability.

It is important to clarify the notion of runtime-related software variability considered in this thesis. Running programs inherently feature manifold degrees of variability based on the very fact that they are running. Programming languages enable control over variability at many levels. For example, a program *variable* is a symbolic name associated to a storage location; the value stored at that memory location can vary while the program executes. A simple "`if(condition)`" statement can be seen as a variation point, as the program will behave in two different ways, based on whether the `condition` is true or false at a given point in time. Polymorphism in object-oriented languages enables operations that behave in various different ways, based on the arguments supplied to them. Our notion of variability fits this general classification but we focus, naturally, on less explored dimensions along which a running program can feature variability, and we look into extending control to the developer over such dimensions. The emphasis in this chapter is on a language for *modelling* runtime variability. The execution of a program can vary according to many conditions. By restricting the variability of a program (along certain dimensions) to a well-defined model, a higher level of trust in its behaviour can be achieved, as this provides a guarantee that the program will not behave in undesired ways.

This chapter documents the framework of ABS language constructs and tools that enable running ABS programs to adapt dynamically. The work in this context was carried out in collaboration with Dave Clarke and José Proença and was published as *Executable Modelling of Dynamic Software Product Lines in the ABS Language* at the Fifth International Workshop on Feature-Oriented Software Development (FOSD 2013) [86]. It builds on the ABS framework for static modelling of software product lines (Chapter 3) and extends it to support models of dynamic SPLs (DSPLs). Section 4.1 explains in brief the concurrency model of ABS, on which our dynamic state update mechanism is based. Section 4.2 introduces DSPL and gives an overview of the ABS extensions that support DSPL development. Section 4.3 details the ABS language elements for modelling dynamic SPLs. Section 4.4 describes auto-reconfiguration using the MetaABS reflective programming interface. Section 4.5 explains how ABS

supports openly adaptive models. Section 4.6 describes the back-end technology that enables the concurrent reconfiguration of systems at runtime. Section 4.7 discusses related work.

4.1 Background: ABS Concurrency Model

The concurrency model of ABS is based on active objects, asynchronous method calls, and futures. Asynchronous method calls trigger concurrent activities, as both the calling and the called methods run in parallel. Futures enable the calling process to later retrieve the result of an asynchronous computation. In ABS, concurrent objects can be collected in concurrent object groups (COGs), which communicate solely via asynchronous method calls [105]. COGs are active runtime entities possessing their own thread; objects inside the same COG share a common scheduler and message queue. Methods execute as tasks inside a COG and use cooperative multitasking, meaning that they release control of the thread only at designated points within the code, using await and suspend statements. An **await**`(guard)` statement causes the process to suspend until the guard is true. Suspension of a process gives another process the opportunity to run. Naturally, control of the thread is relinquished whenever a method finishes.

ABS's concurrency model creates additional challenges when dynamically updating a system. Interrupting the application globally at certain quiescent execution points in order to perform the update is not desirable for reasons of performance. It might not even be practical due the difficulty to ensure that all threads reach quiescence simultaneously. The alternative is to update the system incrementally, that is, to update each active object independently, while the rest of the system continues to run. This approach needs to prevent the system from reaching a globally inconsistent state. In an inconsistent state, messages sent between active objects my not be understood or may result in a call to the incorrect method body being run. The order in which objects are updated is therefore important. If, for instance, an object has a field that references another object, then the other object should be updated first. Mutual and circular references need to be dealt with by updating sets of objects simultaneously.

4.2 From Static to Dynamic Software Product Lines

Software product lines (SPL) [94] have gained significant acceptance in the domain of variable software engineering for their ability to reduce development

time and costs, and improve software quality [114]. Typical SPL approaches, such as the ABS approach presented in Chapter 3, do not focus on dynamic variability: the configuration of products occurs statically at development time. In contrast, dynamic software product lines support the generation of system variants at runtime [15]. In other words, a deployed DSPL system that behaves as a certain product can be *reconfigured* (a.k.a. adapted) to behave as a different, valid product without the need to halt the system, recompile and redeploy. This section details our work on extending ABS to add support for DSPL development.

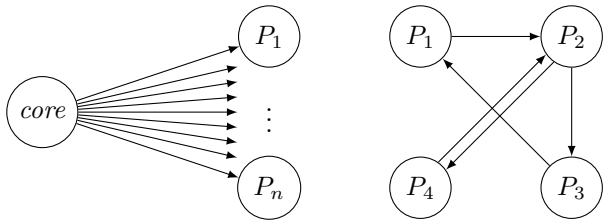


Figure 4.1: Static product configuration transforms a core into any product of the SPL (left). With dynamic product reconfiguration (right), certain transitions between products are allowed at runtime.

To support dynamic adaptation, ABS models need to accommodate runtime changes in their structure and behaviour. Adding this facility to ABS complements its static SPL modelling capability. Static product generation introduced support for configuring a particular SPL product at compile time by taking an ABS *core* model and a set of *deltas* and *flattening* them to obtain an executable ABS model of that single product.

While static and dynamic product configuration are related concepts, they differ in two key aspects (Figure 4.1). Static product configuration always starts with the base product (represented by a *core* ABS model) and applies a sequence of modifications until obtaining any of the products specified by the product line. Dynamic product reconfiguration starts with any product already configured using the above process, and applies a set of modifications to obtain a new product (out of the set of specified products). The second aspect pertaining to dynamic reconfiguration is the necessity to adapt the program's runtime *state* in addition to adapting its structure. The set of products that are configurable from each product at runtime is defined by the ABS developer. The sum of these definitions, together with the information concerning how to modify the product's structure and state upon reconfiguration, effectively describe the runtime variability model.

4.2.1 DSPL Spectrum

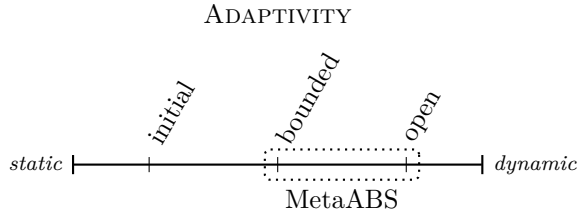


Figure 4.2: DSPLs feature various degrees of dynamicity. MetaABS supports bounded and open adaptivity.

The notion of "dynamic software product line" captures system families with various degrees of dynamicity, depending on the types of runtime adaptation that are supported (Figure 4.2). At the most conservative end, it can simply mean that a system is deployed with all its variants included, and that upon starting the system, a configuration process will adapt the system to a certain variant; no further reconfiguration is possible thereafter. We call this level *initial adaptivity*. A more dynamic approach is to maintain a representation of the SPL to allow the system to be reconfigured during runtime to any variant that the SPL supports. This is referred to as *bounded adaptivity* [15], as the system can only adapt within the statically defined boundaries set by the SPL. A higher yet degree of dynamicity is featured by systems that also allow the SPL itself to change at runtime, thus supporting unanticipated evolution: features, code and product variants can be added or removed arbitrarily. This capability is know as *open adaptivity* [15]. ABS supports, through the MetaABS language extension described in Section 4.5, the development of DSPLs that feature both bounded adaptivity and open adaptivity.

4.2.2 Modelling Runtime Variability

Figure 4.3 compares the ABS language elements used for static and dynamic SPL modelling. To model an SPL that can be configured only at compile-time, the ABS developer specifies a feature model, a selection of products, the SPL configuration, a core, and a set of deltas (as described in Chapter 3). We extend ABS to support dynamic SPLs that can be reconfigured at runtime. To accomplish this step, a few new constructs are needed, while some compile-time elements can be omitted at runtime. First among the new ingredients is a *reconfiguration decision model*, which subsumes static product selection and the SPL configuration, describing how the system will be adapted for each possible

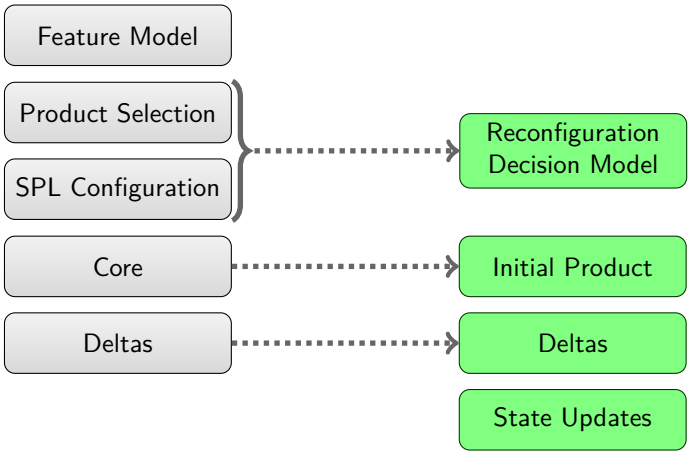


Figure 4.3: ABS static SPL modelling elements (left) and their correspondence to the dynamic SPL modelling elements introduced in this chapter (right)

reconfiguration action. Second, are *state updates*, which define how the system's runtime state will be adapted when its structure changes. State updates and the reconfiguration decision model are described in detail in the following section. The core is only used for static product configuration, of which the result is an *initial product*, that is, the variant in which the system will be deployed. This *initial product* can be later reconfigured dynamically into a different product of the DSPL.

The feature model is used to statically validate product declarations; there is no need to dynamically re-check whether a product is valid, so we forgo to keep it around at runtime. In the case of openly adaptive ABS models, the feature model itself can evolve, for example by adding or removing features and constraints. Hence, the set of products and reconfigurations available at runtime can also change. New products that result from an updated feature model are validated statically and propagated to the running system by updating the reconfiguration decision model.

4.3 Extending ABS for Dynamic Reconfiguration

This section describes the extensions to the ABS language to support dynamic product lines. How these are compiled and the runtime support for them are described in Section 4.6. A DSPL in ABS is a set of software products that are available at runtime, together with a reconfiguration decision model that

describes the variability of the system at runtime as a set of reconfiguration steps. A *reconfiguration* takes place between two products and adapts the current product's structure and state (Figure 4.4).

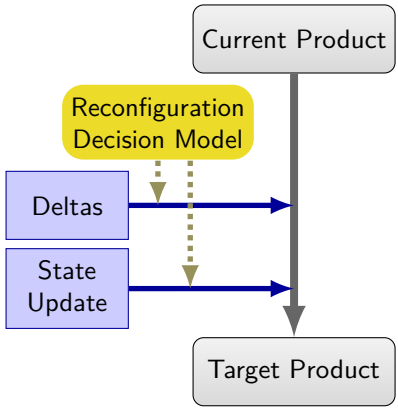


Figure 4.4: Elements involved in reconfiguring a *current* product into a *target* product

The possible reconfigurations between products are specified in a *reconfiguration decision model* (Section 4.3.1). The reconfiguration is performed by dynamically applying *deltas* (Section 4.3.2), and the state of objects is transformed according to state transformation functions declared in a *state update* (Section 4.3.3). As a result of reconfiguration, a different product becomes active and the system behaves according to the specification of the new product. That new product can be adapted into yet another product and so forth. Throughout this chapter we refer to the product that is about to be adapted as the *current* product and the product obtained after adaptation as the *target* product. Reconfiguration of a DSPL can be initiated within the product line using auto-reconfiguration code written in the ABS meta-language, MetaABS (Section 4.4). The section concludes with a comparison of our approach with that of Dynamic Delta-Oriented Programming (Section 4.3.4).

4.3.1 Reconfiguration Decision Model

The reconfiguration decision model defines the possible reconfigurations and how they are carried out. It effectively describes a Labelled Transition System (LTS), with a finite set of nodes representing the products and transitions between nodes representing the possible reconfigurations. Our approach is similar to Damiani and Schaefer [40], who first proposed a reconfiguration automaton that

specifies how to switch between runtime configurations. In their approach, the transitions are labelled with state transfer functions but how the system's classes need to be adapted is left somehow implicit. We discuss several possibilities in Section 4.3.2 and describe the solution implemented by ABS in its concurrent, active objects setting.

A reconfiguration is always performed between two products that are variants of the DSPL. The products are declared, as in the static SPL setting, by associating the product name with a set of features from the feature model. Additionally, each product declaration lists the possible target products of the SPL that the given product can be transformed into, together with a sequence of deltas and state update declarations.

```
1  product LowEnd (Text) {
2      Regular delta DVoice stateupdate L2R;
3  }
4  product Regular (Text, Voice) {
5      HighEnd delta DVideo,DFiles stateupdate R2H;
6      LowEnd  delta DNoVoice stateupdate R2L;
7  }
8  product HighEnd (Text, Voice, Video, Files) {
9      Regular delta DNoFiles,DNoVideo stateupdate H2R;
10 }
```

Figure 4.5: Product declarations for the dynamic chat SPL

The reconfiguration decision model for the chat SPL (Figure 4.5) defines three products, by stating—for each product—the product's name and features, and the set of other product that it can be reconfigured into. The "low-end" chat product (line 1) implements only the `Text` feature. This product can be reconfigured at runtime into a "regular" chat product (declared in line 4) that additionally supports the `Voice` feature by applying the delta `DVoice` and the state transfer function `L2R`. The third product is a "high-end" chat system that also supports video and file transfer (line 8). Both the static configuration options for the chat SPL and its reconfiguration decision model can be readily visualised (Figure 4.6).

Syntax

Figure 4.7 specifies the grammar of the ABS reconfiguration decision modelling language, which extends the product selection language (cf. Section 3.3). The *Selection* clause has a *Reconfiguration* list as an additional element.

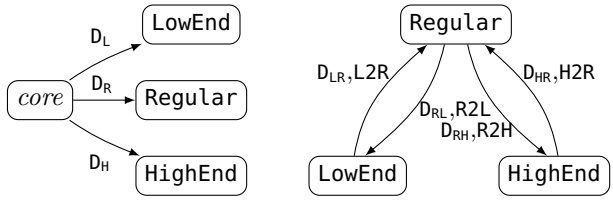


Figure 4.6: Left: static chat product configuration, right: dynamic chat product reconfiguration

Product	::=	`product` *TypeId* `(` *FeatureSpec** `)` `{` *Reconfiguration** `}`
FeatureSpec	::=	FID [*AttributeAssignments*]
AttributeAssignments	::=	`{` *AttributeAssignment* (`,` *AttributeAssignment*)* `}`
AttributeAssignment	::=	AID `=` *Literal*
Reconfiguration	::=	*TypeId* [`delta` *Deltas*] [`stateupdate` *TypeId*] `;`
Deltas	::=	*TypeId* (`,` *TypeId*)*

Figure 4.7: Reconfiguration decision model grammar

Reconfigurations are used to define the valid transitions between products and to specify how the current product will be adapted. A reconfiguration names the target product, then specifies a sequence of delta identifiers and finally assigns a state update to this transition.

4.3.2 Deltas

Transforming a software product with a certain set of features into a product with a different set of features generally requires changing both its structure and behaviour. For an ABS model this entails adding, removing or modifying model elements such as classes, interfaces, functions and data types. As in the static setting (cf. Section 3.4), this is done by applying a sequence of deltas to a core program, except that now deltas are applied while the system is running.

ABS requires the developer to declare the delta sequences necessary for each reconfiguration. For example, the reconfiguration decision model for the Chat product line (Figure 4.5) shows that the `LowEnd` product needs to apply the `DVoice` delta when adapting to the `Regular` product (`with` clause in line 2). In general, this manual approach requires some overhead from the developer, who needs to declare deltas in addition to the deltas used for static reconfiguration

and specify their order of application. For the chat SPL example, the three deltas `DNoVoice`, `DNoVideo` and `DNoFiles` (shown in Figure 4.8) specify the removal of the `Voice`, `Video` and `Files` features. Deltas defined for static product configuration can be also applied dynamically, as is the case when reconfiguring the `Regular` product into the `HighEnd` product by using deltas `DVideo` and `DFiles`.

```
delta DNoVoice;
uses Chat;
removes interface Voice;
modifies interface Client removes Voice;
modifies class ClientImpl {
   removes CallHistory callHistory;
   removes Call call(Client client);
}
removes interface AudioStream;
removes interface Call;
removes class CallImpl;

delta NoVideo; ...
delta NoFiles; ...
```

Figure 4.8: Deltas used in the reconfiguration decision model (Figure 4.5)

There are other approaches to obtaining the deltas necessary for each adaptation. Rather than having the developer specify them, deltas between pairs of products could be also inferred, addressing possible issues of scalability with large product lines, which would otherwise require the definition of a complex reconfiguration decision model. One alternative is to statically construct the delta based on the difference between the current and target product. This implies that the entire set of products available at runtime needs to be statically generated. Then, a delta can be inferred for each product transition, based on the difference between each product pair. A further approach is based on deriving a delta from the existing deltas used for static product configuration. This involves consolidating the modifications described by the sequence of static deltas following the approach described by Clarke et al. [28] together with calculating inverse deltas for reversing these modifications. The approach taken here, of requiring the developer to specify the deltas, gives the developer more control over reconfiguration.

4.3.3 State Updates

A running system has an execution state, that is, the collection of values assigned to variables and fields, which is commonly stored on a stack and heap. When reconfiguring a running system, these values need to be preserved or adapted to the system's new structure. The challenges in this context are related to how to adapt state elements to match the updated system, and to when to adapt state elements without disrupting the runtime execution.

While fully automated state update has been the focus of recent research [51, 82], the transfer of state information typically requires some manual guidance from the developer in cases when state variables need to be mapped to new variables by a function more complex that simple identity. For example, if a field is removed, its value might need to be preserved by transferring it to a new field of possibly different type. Similarly, if a new field is added, it needs to be given sensible value, which may not be the default value.

We adopt a hybrid approach, automating the simple cases (e.g., fields that are present in the old and new code are carried over unaltered), combined with the ability for the user to manually define the transfer function for more complex scenarios. An ABS *state update* specifies *how* to transfer the values of fields to the object's post-reconfiguration state while also mandating *when* it is safe to do so. A state update declaration typically hosts the state transfer functions between a certain pair of software products. State updates are connected to reconfiguration transitions in the reconfiguration decision model.

A state update is a collection of *object updates*, each describing how to update objects from a given class. More precisely, an object update consists of: (1) the name of the class whose instances are targeted by the update, (2) an *update guard* mandating when the state update can be applied, (3) a set of declarations of local variables and functions used within the body of the object update, (4) a **classupdate** statement that triggers the update of the object's class (thus updating its interface and fields), and (5) a set of assignments used to initialise the object's fields, possibly based on values of its state before updating the class. When an object update is applied to update an object, the following steps are performed:

1. Wait until the update guard becomes true
2. Initialise any update-local variables
3. Update the object's class pointer to new class version
4. Initialise the object's new state based on update-local variables

The update guard is expressed by an ABS **await** statement, allowing the developer to specify when it is safe to apply the state update in a concurrent system. The code before the **classupdate** is run in the context of the object's original type and is used to salvage values of fields that are removed by the update. The code after the **classupdate** is used to initialise added fields, possibly with values computed in the pre-update step. Note that the values of fields that are not affected by the state update (i.e. they are present in the old and new state) are carried over automatically. The names in scope before and after the **classupdate** are thus different. The scope changes according to how the object's class structure changes. For instance, if fields are removed, their names are not available after the class update. Similarly, if new fields are introduced by the new version of the class, they are only available after the class update. Variables defined locally within the object update bridge these two scopes.

Applying an object update triggers the creation of a task that is scheduled to be executed on the COG where the object resides. This task can execute only when the update guard becomes true. The details of the implementation will be described in Section 4.6.1.

```
stateupdate R2L;
objectupdate ClientImpl {
    // pre–update section
    await(length(ongoingCalls == 0));

    def ChatHistory mergeHistories(CallHistory calls,ChatHistory chats)=...
    ChatHistory mergedHistory;
    mergedHistory = mergeHistories(callHistory,chatHistory);

    classupdate; // change in scope
    // post–update section
    chatHistory = mergedHistory;
}
```

Figure 4.9: State update example

An example state update (Figure 4.9) is used to adapt a "regular" chat software to the "low-end" variant. The update of chat clients (objects of class `ClientImpl`) is guarded by a condition that requires that no calls are ongoing with the client involved. We assume that the state of the regular client includes a history of calls and a list of chat sessions, stored in the fields `callHistory` and `chatHistory`, respectively. The `chatHistory` is common to both products, therefore its value is preserved by default. However, the history of calls will be lost upon removing the `callHistory` field, as directed by the delta `DNoVoice` (cf. Figure 4.8). To keep

this information, both histories are merged into in a local variable `mergedHistory` in the pre-adaptation phase. In the post-adaptation phase this value is assigned to the `chatHistory` field.

Syntax

Figure 4.10 shows the grammar of state updates. A state update has a name handle and a list of *ObjectUpdate*s that specify how the state of objects of a certain class should be transferred upon reconfiguration. An *AwaitStmt* defines the update guard. A set of locally scoped variables, data types and functions can be declared thereafter. *Statements* before the `classupdate` keyword run in the context of the object's original type, while *Statements* thereafter are executed after the class adaptation has been performed.

StateUpdate	::=	`stateupdate` *TypeId* `;` *ObjectUpdate**
ObjectUpdate	::=	`objectupdate` *TypeId* `{` *Body* `}`
Body	::=	*AwaitStmt Decl** *Statements* `classupdate;` *Statements*
AwaitStmt	::=	`await` *Guard* `;`
Decl	::=	*VarDecl* | *DataTypeDecl* | *FunctionDecl*
Statements	::=	*Statement**

Figure 4.10: State update grammar

4.3.4 Comparison with Dynamic DOP

Our approach is based on Dynamic DOP [39, 40], but there are two key differences. The first difference is in the underlying object/concurrency model in the core language. Dynamic DOP is based on Java's single threaded objects, whereas ABS is based on active objects. This has an impact on how reconfiguration can be performed. One advantage of an active object-based setting is that updates can be performed incrementally, per object, without stopping the entire system.

The second difference is that reconfiguration in Dynamic DOP is triggered using a `reconfigure` statement. This statement can occur anywhere within the body of a method. When executed, it triggers a reconfiguration to be performed, if one is pending. Reconfiguration is thus initiated from within the code, which requires the programmer to anticipate possible future reconfiguration. In contrast, in the reconfiguration model presented here, the trigger of updates is specified externally to the code being updated and there is much more control

over when updates are made, as each update awaits each object being updated to reach a particular state. Thus updates can be applied more flexibly.

A version of Dynamic DOP's `reconfigure` can be encoded in our model. The difference with this version is that it allows reconfiguration to be done on a per object basis, rather than globally. One downside of this is that only the objects calling `reconfigure` can be reconfigured. The Dynamic DOP code in Figure 4.11 (written using ABS syntax) uses `reconfigure` to trigger reconfiguration.

```
class Controller {
   Unit run() {
      ...
      reconfigure;
      ...
   }
}
stateupdate BonusAccount2BasicAccount;
objectupdate Account {
   Int tmp1 = this.balance; Int tmp2 = this.bonus;
   classupdate;
   this.balance = tmp1 + tmp2;
}
```

Figure 4.11: Dynamic DOP reconfiguration example

This can be encoded in ABS as shown in Figure 4.12. In this code, when **suspend** is executed within an active object, control of the thread is released, allowing another thread to be scheduled within the COG. This thread initiates the update of this object, depending upon the scheduler's policy.

4.4 MetaABS Support for Auto-Adaptation

To support product line adaptation autonomously at runtime, ABS introduces a dynamic meta-programming facility, called MetaABS, based on reflection. MetaABS exposes basic elements of the programming language, and the runtime environment to the programmer, enabling their inspection and modification. Among these elements are the running system's underlying DSPL structure. This gives the running system the capability to reconfigure itself.

```
class Controller {
   Bool __reconfigurable = False;
   Unit run() {
      ...
      __reconfigurable = True;
      suspend;
      __reconfigurable = False;
      ...
   }
}
stateupdate BonusAccount2BasicAccount;
objectupdate Account {
   ABSRuntime.await(__reconfigurable);
   Int tmp1 = this.balance; Int tmp2 = this.bonus;
   classupdate;
   this.balance = tmp1 + tmp2;
}
```

Figure 4.12: ABS encoding of Dynamic DOP reconfiguration example

4.4.1 Metaprogramming

Metaprogramming is generally understood as the ability to observe and modify the structure and behaviour of a program from within a program, either statically or at runtime. A metaprogramming interface exposes basic elements of the programming language and the runtime environment to the programmer, enabling their inspection and modification. While it exposes these elements, it also abstracts away from their implementation.

Languages that support metaprogramming commonly achieve this by providing *reflection*, that is, the ability of a program to inspect and modify itself at runtime. Thus the metaprogram (the program transforming program) and the program that is transformed are the same. Reflection is decomposed into *introspection*, meaning the ability of a program to examine itself, and *intercession*, which enables a program to modify its state and behaviour. In other words, introspection and intercession provide, respectively, read and write access to elements of the language. For example, the Java Reflection API is a metaprogramming interface that provides methods to examine, and, to a very limited extent, modify the runtime properties of objects including their class, interfaces, fields and methods.

Systems that adapt their behaviour at runtime often need to do so autonomously, by monitoring certain variables in their operating environment and adapting as

these variables change. To allow ABS systems to self-adapt autonomously, we extended ABS with a reflective metaprogramming interface that exposes basic elements of the programming language, and the runtime environment to the programmer, enabling their inspection and modification.

4.4.2 Motivation and Scope of MetaABS

Being first and foremost a research vehicle, the ABS language is frequently used as a basis for implementing language-based program analysis tools. Program aspects of particular interest within the HATS project include the scheduling of tasks inside concurrent object groups; the dynamic reconfiguration of software products; deployment component configuration; and runtime method dispatch. To aid this this situation, MetaABS is intended as a unified, general-purpose meta-interface that can reflect on any desired aspect of the ABS model being executed. Thus the scope of MetaABS is *runtime* program analysis and configuration. The motivation for providing such an interface was to avoid the need to extend the ABS language itself (specifically its grammar) with new constructs for each specific analysis task. For example, the inclusion of a component model in ABS [78] introduced six new keywords to the ABS grammar along with six new types of non-terminals. Some ABS extensions (such as real-time parameters [20] and deployment components [71]) have been implemented using *annotations*. Annotations are a standardised form of expressions enclosed in square brackets that are written next to program statements to configure certain details of their behaviour. While this method is less invasive than defining new keywords and non-terminals, it still requires to change the grammar to allow annotations at specific places in the code.

MetaABS exposes internals of ABS models, such as classes, methods, the variability model, object state and the scheduling of processes to the programmer, enabling their inspection and modification. While it exposes these elements, it also abstracts away from their implementation, providing a clear, easy to use object-oriented model. Having access to these *metaobjects* makes it possible to analyse a model while it is executed.

4.4.3 MetaABS

MetaABS is an object-oriented reflective interface to the ABS language. It provides an abstraction of the underlying ABS runtime, making it independent from any actual implementation. MetaABS is implemented as a library alongside the ABS standard library. It is easily extensible should new requirements arise. Figure 4.13 shows the interface structure of MetaABS. In this section we only

detail the types and operations that are related to inspecting and changing the DSPL structure and configuration. The MetaABS API is designed around the concept of mirrors [23]. By using mirror interfaces to access the behaviour of metaobjects we achieve a clear separation between the behaviour of objects (determined by their type) and their mirrors.

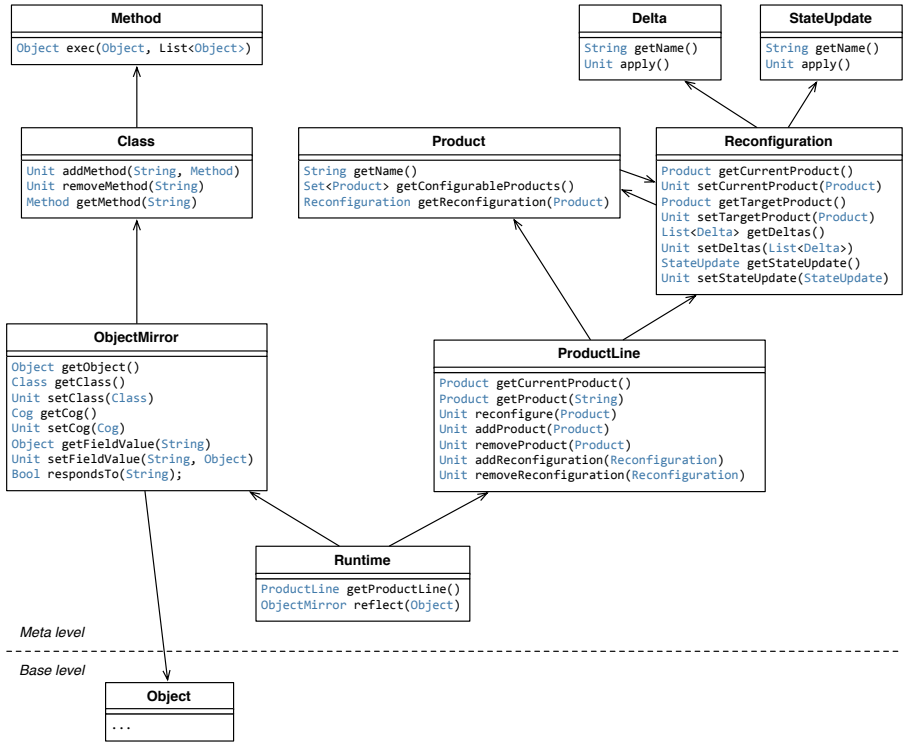


Figure 4.13: The MetaABS API reflecting on the DSPL

The Runtime

The `Runtime` interface is the entry point to MetaABS. A `Runtime` object is obtained by instantiating the `Runtime` class provided by the `ABS.Meta` library. The following code demonstrates the usage.

```
module Test;
import * from ABS.Meta;
{
   Runtime rt = new Runtime();
   ProductLine pl = rt.getProductLine();
   Product p = pl.getCurrentProduct();
}
```

MetaABS Types Reflecting on Objects

Object Mirrors An `ObjectMirror` reflects on an existing ABS object. An object mirror is obtained by calling the built-in function `reflect(object)` on any given ABS object.[1] The object mirror provides a set of reflective operations such as for getting or setting the object's class and its concurrent group affiliation (cf. Figure 4.13). The following example illustrates how reflective operations are accessed from an object mirror.

```
import * from ABS.Meta;
class C implements I { Unit foo() {...} }
{
  I obj = new C();
  ObjectMirror mir = reflect(obj);
  Class cls = mir.getClass();
  cls.removeMethod("foo");
}
```

Object `Object` is the type of ABS objects. One can use reflective operations on objects by first using the function `reflect(object)` and then calling a reflective operation on the returned `ObjectMirror`. `ObjectMirror` provides a `getObject()` method that returns the `Object` it reflects upon.

Classes A `Class` type represents an ABS class. Its interface includes operations to add and remove methods.

[1] In the current implementation, the `reflect` operation is not accessible through the `Runtime` interface, but rather through a dedicated function. This because ABS currently does not support generic methods, only parametric functions; it is therefore not possible to define a method that applies to any type of object.

MetaABS Types Reflecting on the DSPL

As shown in the previous section, ABS provides extensive support for modelling dynamic software product lines. A section of MetaABS is therefore dedicated to reflecting the DSPL structure and configuration back to the ABS program, thus enabling dynamic auto-reconfiguration. For this purpose, the DSPL, i.e., its reconfiguration decision model, deltas and state updates, are made accessible as objects of the language, each with their own set of MetaABS operations.

An ABS model developed as a software product line behaves at runtime as one particular product defined by the SPL. MetaABS operations make it possible to reconfigure this product into a different product of the DSPL while the system is executing.

Product Line The `ProductLine` is an interface to the DSPL that governs the running ABS system. By calling `getProductLine` on the `Runtime` object, an object of type `ProductLine` is obtained. Its interface provides ABS methods to query the current product and the products that can be obtained through reconfiguration of the current product, as defined by the reconfiguration decision model (Section 4.3.1). Beyond simple introspection, it provides a `reconfigure(Product)` operation to actually trigger the reconfiguration of the current system to behave as a given target product.

Product The `Product` type represents program variants defined by the reconfiguration decision model of the DSPL. The runtime `Product` interface exposes the possible transitions to other products through the `getConfigurableProducts` operation.

Reconfiguration A `Reconfiguration` connects two products (the current and the target product), associating a list of deltas and a state update to the transition between the them. The `getDeltas` and `getStateUpdate` operations provide the list of deltas and the state update that need to be applied in order to reconfigure the current product into the target product. Products and reconfigurations together represent the reconfiguration decision model defined in ABS (Section 4.3.1).

Delta The `Delta` type is used to represent deltas (cf. Section 3.4). The principal operation on deltas is `apply`, meaning that the modification operations defined by the delta will be applied to the current program. Applying a delta at runtime is different from applying a delta statically, where the delta modifiers

directly change the program's AST (cf. Section 3.4). Runtime delta application is performed in two steps to ensure the program continues to execute normally. In the first step, the delta modifiers are applied to copies of the classes that they modify. The second step–replacing the old classes with their updated versions–is performed incrementally, that is, individually for each object, along with adapting its state. The runtime application of deltas is described in detail in Section 4.6.1.

State Update The `StateUpdate` type represents state updates (cf. Section 4.3.3). A state update applies to the set of objects in the system and results in the transformation of the program's state to match its new structure obtained through the application of deltas. Each object is updated individually when its quiescence condition is met. The runtime updating of object state is described in detail in Section 4.6.1.

4.4.4 Example

An example use of MetaABS to implement the global triggering of reconfiguration of the chat dynamic product line is now given. This example (Figure 4.14) models a system that adapts its behaviour autonomously by monitoring certain variables in its operating environment and adapting as these variables change. The reconfiguration logic is encapsulated in a `Reconfigurator` class. A reconfigurator instance runs as a separate process (that is, concurrently to the chat functionality), monitors the network connection and transforms the running product depending on the available bandwidth. The highlighted calls invoke methods provided in the `ABS.Meta` library.

4.5 Open Adaptivity

DSPL that can evolve at runtime by incorporating changes into their variability model are *openly adaptive* [15]; put differently, they support meta-variability. Such changes can be the addition, removal or modification of products or transitions between products. ABS supports open adaptivity by allowing changes to the reconfiguration decision model via the MetaABS language. The only restriction we impose is that the currently running product cannot be removed when the reconfiguration decision model is updated. Should this become necessary (for example if the evolved set of products is disjoint from the current set), then the adaptation has to be performed in two steps. The first step would add the new products, along with a reconfiguration from the current

```
module Monitor;
import * from ABS.Meta;
data Bandwidth = Low | Mid | High;
interface Connection { Bandwidth checkBandwidth(); }
class Reconfigurator(Connection conn) {
   Unit run() {
      ProductLine pl = getProductLine();
      while(True) {
         Product currentP = pl.getCurrentProduct();
         Product targetP;
         Bandwidth bw = conn.checkBandwidth();
         if (currentP.getName() == "RegularChat") {
            if (bw == Low) {
               targetP = pl.getProduct("LowEndChat");
               pl.reconfigure(targetP));
            } else if (bw == High) {
               targetP = pl.getProduct("HighEndChat");
               pl.reconfigure(targetP));
            }
         } else if (p == "HighEndChat") {
            if (bw == Low || bw == Mid) {
               targetP = pl.getProduct("RegularChat");
               pl.reconfigure(targetP));
            }
         } else if (p == "LowEndChat") {
            if (bw == Mid || bw == High) {
               targetP = pl.getProduct("RegularChat");
               pl.reconfigure(targetP));
            }
         }
      }
   }
}
```

Figure 4.14: Implementing runtime auto-reconfiguration of the chat product line using MetaABS

product into one of the new products. After this reconfiguration is performed, the old products can be safely removed in the scond step.

Adding variants to an existing DSPL requires injecting the corresponding code (i.e. products, reconfigurations, deltas and state updates) into the system at runtime. Our system provides a runtime interface that allows dynamic loading of code via Java's standard class loading mechanism.

The MetaABS API (introduced in Section 4.4) provides operations to add and remove products as well as reconfigurations, that is, transitions between products. It also supports modifying existing products by adding or removing reconfigurations, and re-setting a reconfiguration's state update and delta sequence.

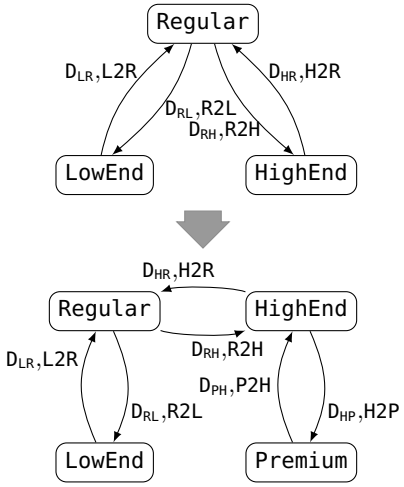


Figure 4.15: Evolving the chat DSPL

The model a programmer should have in mind when thinking about open adaptivity is that of a transformation between reconfiguration decision models. For instance, Figure 4.15 shows how the chat SPL could evolve by adding a `Premium` product that implements video conferencing. The implementation of this evolutionary step is provided in Figure 4.16.

Open adaptivity is currently supported by the ABS runtime and available to the programmer via the MetaABS interface. A quite natural extension to this mechanism is to allow the user to *model* the adaptation of the reconfiguration decision model, i.e, to define a meta-variability model. To stay within the delta-oriented paradigm, deltas could be used to define modifications to the reconfiguration decision model. In this case, the code shown in Figure 4.16 would be *generated* from a delta declaration.

4.6 Dynamic Reconfiguration

Dynamic software product reconfiguration has two phases: changing the system structure and behaviour by applying deltas, and subsequently mapping the

```
ProductLine pl = getProductLine();
Product high = pl.getProduct("HighEnd");
Product prem = new Product("Premium");
pl.addProduct(prem);
Reconfiguration h2p = new Reconfiguration();
h2p.setCurrentProduct(high);
h2p.setTargetProduct(prem);
h2p.setDeltas(...); h2p.setUpdate(...);
Reconfiguration p2h = new Reconfiguration();
p2h.setCurrentProduct(prem);
p2h.setTargetProduct(high);
p2h.setDeltas(...); p2h.setUpdate(...);
pl.addReconfiguration(h2p);
pl.addReconfiguration(p2h);
```

Figure 4.16: Adapting the reconfiguration decision model of the chat DSPL: MetaABS implementation

old execution state to the new system structure. The ABS user controls the application of both deltas and state updates via the reconfiguration decision model. This section describes the mechanisms that enable dynamic reconfiguration in ABS, that is, the back-end mechanisms that delta and and state update application rely upon.

The adaptation of a running program requires a supporting runtime environment. We designed an adaptive runtime environment and an ABS compiler back-end that generates adaptive Java code. We refer to this tool infrastructure as the *dynamic Java back-end*. The dynamic Java back-end is implemented and available[2] as part of the ABS tool framework (cf. Figure 3.18).

The key idea behind the dynamic Java back-end is to use dynamic structures in the target language to represent ABS elements. Whereas the standard Java back-end represents ABS classes, functions and data types as Java classes and ABS interfaces as Java interfaces, the dynamic Java back-end uses Java objects created using the singleton design pattern to represent ABS elements. This applies to core ABS elements, i.e., interfaces, classes, methods, objects, object fields, cogs, data types, functions etc., as well as elements of the variability model, such as the SPL, products, reconfigurations, deltas and state updates. An example shall illustrate this setup. Adding a new class to the system is a common activity when configuring a new product. The new class is represented as an instance of the class `ABSDynamicClass`, which is provided by the runtime

[2] `http://tools.hats-project.eu/spl/dynamic.html`

environment. Fields and methods of the new class are also encoded as objects, and are associated with the class by calling the `addField` and `addMethod` on the class instance. Modifying an existing class amounts to adding and removing fields and methods by calling `addField`, `addMethod`, `removeField` and `removeMethod`. Such a representation trades execution performance for fully malleable ABS models.

4.6.1 Concurrent Reconfiguration

We designed the runtime updating mechanism to use ABS's own concurrency model, according to which the methods of a group of objects are scheduled as tasks in response to asynchronous method calls. Following the cooperative multitasking scheme, tasks can suspend voluntarily. Like methods, updates are scheduled as tasks, in response to system reconfiguration requests (typically the MetaABS operation `Product.reconfigure`). In order to control when updates are applied to specific objects, a standard **await**`(guard)` statement is used. The guard defines the quiescence condition for each object. As long as the condition is false, the update task suspends; when true, the update is executed to completion. This gives the ABS developer the power to formulate what is considered a safe state for the objects of each class, and it ensures the update is performed when the object is in such a state.

In the following, our concurrent reconfiguration mechanism for ABS is presented in detail. Reconfiguring a running ABS system corresponds to globally modifying the system's class structure by applying a sequence of deltas, and incrementally updating each object by applying a state update (cf. Section 4.6).

The Object Roster A state update contains *object updates* that define the reconfiguration of objects of a certain class. Prior to scheduling an object update, the system needs to be obtain the set of objects in the system to which the object update applies. For this purpose, the runtime maintains the object roster, a set of objects for each class. In our implementation the roster has weak references, allowing the JVM garbage collector to collect objects that are no longer in use. The object roster is cleared periodically of references to objects that have been collected.

Sequencing Object Updates Object updates are applied in the order in which they were deployed, that is, no update can overtake one that was triggered by an earlier reconfiguration request. To ensure this, updates and objects bear version numbers. To apply an update to an object, their versions must be equal.

Objects increment their version number after having been updated. Technically, if an update process is scheduled to run and the versions do not match, the process suspends. Consequently, the next-in-sequence update process gets a chance to run.

Global Reconfiguration Scheme

A product is reconfigured in two steps, as illustrated by the algorithm in Figure 4.17. First, a sequence of dynamic deltas `D1..Dn` is applied to a copy of the targeted classes, extending the system's class structure. These new classes will be linked to the running objects in the second step. Secondly, all objects affected by structural changes (that is, all objects whose class was modified) are scheduled for update. The application of deltas and updating of objects are performed using MetaABS operations.

```
Require: deltas D1..Dn; state update U
  for all deltas D in D1..Dn do
    D.apply()
  end for
  for all classes C targeted in state update U do
    for all instances obj of class C do
      ObjectMirror objm = reflect(obj)
      ObjectUpdate u = U.getUpdate(C)
      objm.scheduleUpdate(u)
    end for
  end for
```

Figure 4.17: Global reconfiguration schematic

The first `for` loop in Figure 4.17 shows the application of the delta sequence associated with a reconfiguration. Deltas are applied atomically for each class. To achieve atomicity, all class modifications are applied to a copy of the targeted class; after a class copy is created and the modifications are applied, the copied class retains a pointer to its successor through a `nextVersion` field. The successor eventually replaces the original class. When a delta adds a class, the new class is created. In case of class removal the targeted class is marked for removal (which prevents creating new instances), but it is only removed when no more objects of that class exist in the system. The timing of delta application is therefore not critical.

Object Updating Scheme

Object updates are scheduled individually for each object as tasks of their respective COG. They are executed observing their application guards. This corresponds to the second `for` loop in the reconfiguration scheme (Figure 4.17). The object updating algorithm is broken down into individual MetaABS instructions as shown in Figure 4.18. Finally, objects that are not targeted by the state update but whose classes have been updated by a delta need to update their class reference to the new class version. For these objects, the compiler generates object updates implicitly, with empty field transformations, and guard set to `True`; they are included in state update `U`.

Require: new class `C`; state update `U`
  await(**this**.version == U.version && U.guard)
  initialise local definitions
  run pre-**classupdate** body of `U`
  `Class c =` **this**`.getClass()`
  `Class newC = c.getNextVersion()`
  this`.setClass(newC)`
  run post-**classupdate** body of `U`
  this`.version += 1`

Figure 4.18: Per-object reconfiguration schematic

Object updates are executed after a user-defined condition (`guard`) becomes `True` and the object matches the version of the update. The application of an update consists in first saving elements of the old state that need to be preserved, then updating the object's class pointer to the new class version (created through delta application), and then initialising the new state, possibly based on values saved from the old state. Finally, the object's version is incremented.

The `R2L` state update (Figure 4.9), for example, is converted into the task presented in Figure 4.20. This example (arbitrarily) assumes that the update's version number is 2. The task is scheduled for every instance o of `ClientImp`, in the same way if it was a method of o being invoked asynchronously.

4.6.2 Dynamic Java Back-end Design

Figure 4.21 shows the overall design of the dynamic Java back-end. When compiling an ABS model to Java using the dynamic Java back-end, MetaABS types and operations (Figure 4.13) are mapped to this Java interface.

```
stateupdate R2L;
objectupdate ClientImpl {
    await(length(ongoingCalls == 0));
    def ChatHistory mergeHistories(CallHistory calls,ChatHistory chats)=...
    ChatHistory mergedHistory;
    mergedHistory = mergeHistories(callHistory,chatHistory);
    classupdate;
    chatHistory = mergedHistory;
}
```

Figure 4.19: State update example (repeated from Figure 4.9)

```
{
  await(this.version == 2 && length(ongoingCalls == 0));

  // pre:
  mergedHistory = mergeHistories(callHistory,chatHistory);

  // class update
  Class c = this.getClass();
  Class newC = c.getNextVersion();
  this.setClass(newC);

  // post:
  chatHistory = mergedHistory;

  // version update
  this.version += 1;
}
```

Figure 4.20: Task performing an object update

Objects, Classes, Methods and Fields

Most ABS language elements are represented by the adaptive runtime environment as instances of class `ABSDynamicObject`, thus making them easily accessible as regular ABS objects. The `ABSDynamicObject` interface allows us to modify an object's class and COG associations, update field values and dispatch messages (method calls) to the appropriate method bodies. An `ABSDynamicObject` has a reference to an `ABSDynamicClass` that defines the object's structure and behaviour by providing fields and methods. The class object also provides a standard set of operations for setting or modifying the class

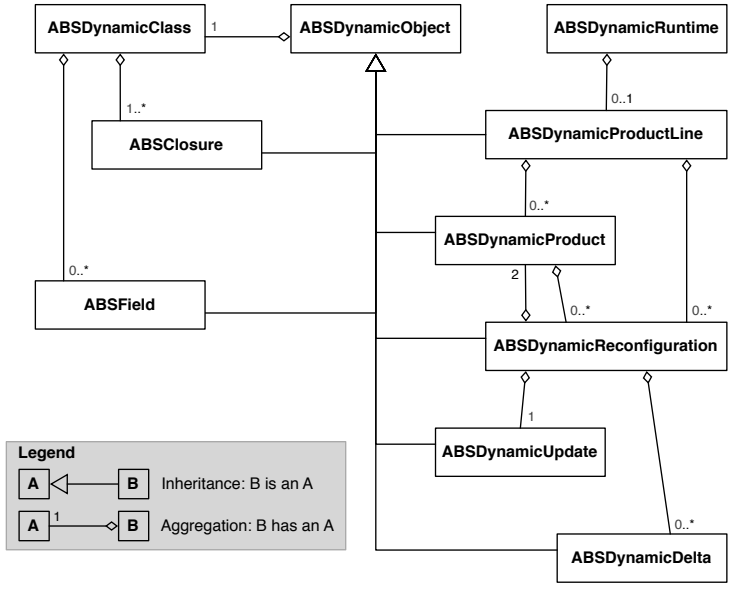


Figure 4.21: Dynamic Java back-end: Java structure

name, initialisation block (constructor), methods, fields and class parameters. These operations are essential in our setting that allows classes and objects to change dynamically. Concrete fields inherit from `ABSField` and provide a specific initialisation expression by overriding the `init` method. Field values are specific to individual objects and thus stored with the corresponding instance of `ABSDynamicObject`. Methods and constructors of classes are represented as objects of type `ABSClosure`. `ABSClosure` is an abstract class whose `exec` method serves as a placeholder for a method's specific behaviour. To create a method, a concrete subclass of `ABSClosure` overriding `exec` needs to be provided.

DSPL Products, Reconfigurations, Deltas and Updates

The DSPL structure of a running ABS system is mapped to a set of interfaces corresponding to elements used to describe DSPLs in ABS: products, reconfigurations, state updates and deltas. The `ABSDynamicProductLine` represents the DSPL as a whole; it implements the `ProductLine` MetaABS interface. The `ABSDynamicProductLine` references a set of products and a set of reconfigurations. These sets can be modified dynamically, thus supporting meta-variable ABS systems, in which the variability mode itself can evolve over time. The `ABSDynamicProduct` represents an SPL product available at runtime.

A product refers to `ABSDynamicReconfiguration`s for the applicable sequence of deltas (instances of `ABSDynamicDelta`) and a state update (`ABSDynamicUpdate`) that dynamically transforms the product into a different product. A concrete `ABSDynamicDelta` prescribes modifications to existing classes, that is, instances of `ABSDynamicClass`, by adding, removing or modifying methods (instances of `ABSClosure`) and fields (`ABSfield`). Similarly an `ABSDynamicDelta` can be used to remove existing classes, effectively marking existing instances of `ABSDynamicClass` for removal, or to add new classes, which effectively creates new instances of `ABSDynamicClass`.

In the following section we provide examples of ABS code elements and their generated representations using the dynamic Java back-end entities introduced in this section.

4.6.3 Code Generation

To illustrate the code generation process for the dynamic ABS Java back-end, we show how a few simple ABS code snippets compile to Java using the dynamic Java back-end, and compare it to the code generated for the regular, static Java back-end (when applicable).

To keep the generated code snippets small we use the following artificial ABS example, illustrated in Figure 4.22:

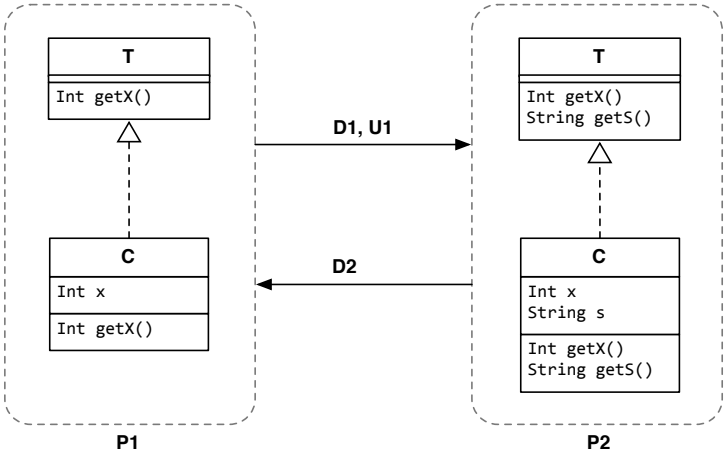


Figure 4.22: A simple DSPL example

The example system contains a class `C` that implements an interface `T` that provides a method `Int getX()`. Instances of `C` use the field `x` to store a value of type `Int`. This system corresponds to an initial product `P1` that can be reconfigured dynamically to a product `P2` by applying a delta `D1` and a state update `U1`. `P2`, in turn, can be reconfigured back into `P1` by applying `D2`; no state update is necessary. Delta `D1` adds a method `String getS()` and a field `String s`; `D2` removes these two elements. The update `U1` transfers the value of `Int x` to `String s`.

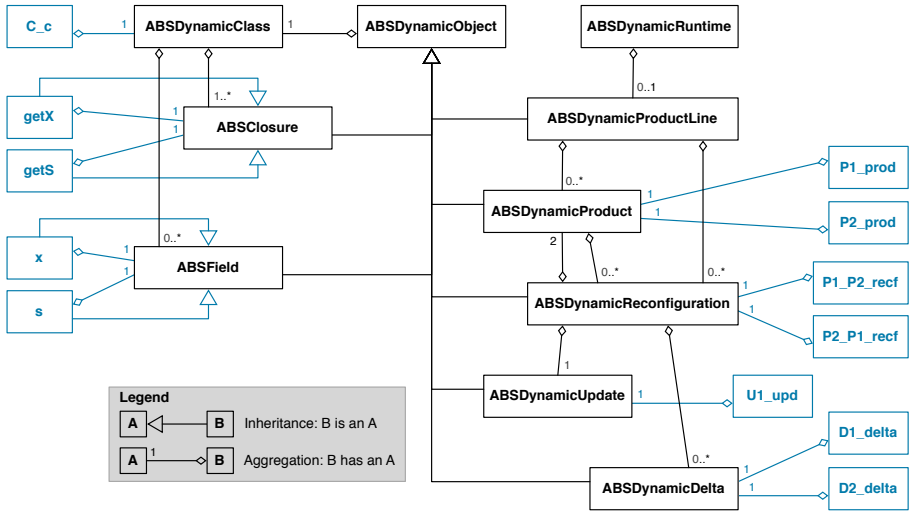


Figure 4.23: The blue classes are generated by compiling the example DSPL; they use the structure provided by the dynamic Java back-end (cf. Figure 4.21).

When compiling this ABS program using the dynamic Java back-end, its elements are mapped to the class structure shown in Figure 4.23. The generated Java classes plug into the dynamic Java back-end infrastructure presented in Section 4.6.2. Figure 4.24 shows the object structure at runtime after the reconfiguration from `P1` to `P2`. Two versions of class `C` exist (0 and 1) At the top left are instances of `C`. Some of these instances still point to version 0 of the class, while others have been already updated to point to the new version 1.

Classes, Methods and Fields

The example in Figure 4.25 illustrates the code generation process for class declarations. The generated static Java code is very similar to the original ABS code: the generated Java class `C_c` corresponds to ABS class `C`, and

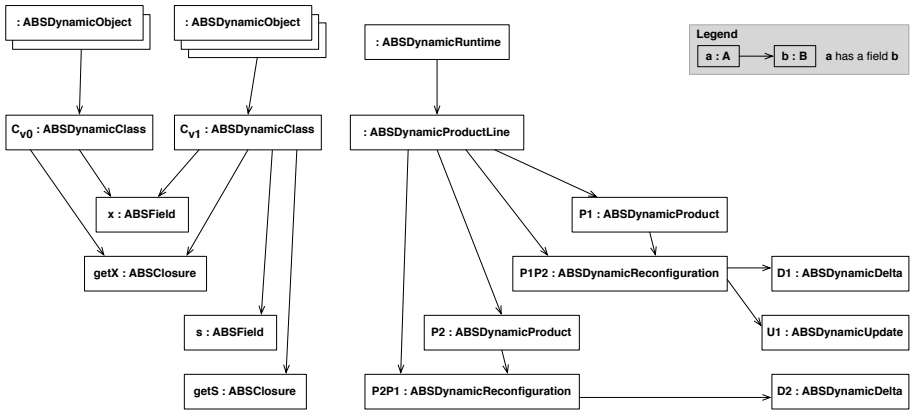


Figure 4.24: Object structure at runtime (product `P2`)

the generated method `ABSInteger getX()` corresponds to `Int getX()`. In the dynamic setting, ABS classes are represented as singleton instances [47] of class `ABSDynamicClass`. The static method `C_c.singleton` (line 3) creates an `ABSDynamicClass` object (line 5) and adds class `C`'s methods and fields and fields. These are represented as subclasses of `ABSClosure` and `ABSField`. For the method `getX`, a new class inheriting from `ABSClosure` is created, which, by overriding the `exec` method, encodes the method's specific behaviour. Similarly, for the field `x`, `ABSField` is subclassed and its `init` method is overridden with the field's specific initialisation expression. Instances of these two classes are passed as arguments to `addMethod` (line 6) and `addField` (line 7).

Objects

The example in Figure 4.26 shows object creation and method calling. In the static setting code generation is straightforward. In the dynamic setting, an object of the predefined class `ABSDynamicObject` is created with a reference to the `ABSDynamicClass` object representing class `C` (line 1). Calling `C`'s method then amounts to calling the `dispatch` method on the `ABSDynamicObject` with the name of the method as an argument (line 2). The `dispatch` method returns a generic `ABSValue` that needs to be cast to the method's specific return type.

ABS Source Code

```
class C(Int x) implements T {
   Int getX() { return x; }
}
```

Generated Java Code

```
public final class C_c extends ABSObject implements ABSClass, T_i {
   private ABSInteger x;
   public C_c(ABSInteger) {...}
   public final ABSInteger getX() {...}
}
```

Generated Dynamic Java Code

```
1  public final class C_c {
2     private static ABSDynamicClass c;
3     public static ABSDynamicClass singleton() {
4        if (c == null) {
5           c = new ABSDynamicClass();
6           c.addMethod("getX", new ABSClosure() { /* Override exec */ });
7           c.addField("x", new ABSField() { /* Override init */ });
8        }
9        return c;
10    }
11 }
```

Figure 4.25: Class declaration encoded using the dynamic Java back-end

Dynamic Software Product Lines

Recall from Section 4.3 that a DSPL is a set of software systems that can be configured dynamically according to a variability model attached to it. To support dynamic product (re)configuration, a runtime representation of the variability model is needed. Dynamic variability in ABS is represented by a a reconfiguration decision model (i.e. a set of products and the reconfigurations that are possible between them), connected to deltas and state updates. This section describes how these elements are represented by the dynamic Java back-end. Note that there is no equivalent representation in generated static Java code, as the static Java back-end uses deltas to modify the AST in place and discards them prior to code generation.

Products The code generation process for ABS products is illustrated in Figure 4.27, based on the product `P1` from the reconfiguration decision

ABS Source Code

```
C object = new C(42);
Int x = object.getX();
```

Generated Java Code

```
C_c object = new C_c(42);
ABSInteger x = object.getX();
```

Generated Dynamic Java Code

```
1 ABSDynamicObject object = new ABSDynamicObject(C_c.singleton(), 42);
2 ABSInteger x = (ABSInteger)object.dispatch("getX");
```

Figure 4.26: Class instantiation and method calling in the dynamic Java back-end

model of the example SPL. Products are represented as Java objects of type `ABSDynamicProduct` (line 6). A product is configured by setting its name (line 7) and recording its valid reconfiguration transitions to other products. Transforming product `P1` into the `P2` product involves applying the reconfiguration `P1_P2` (line 8–9). How reconfigurations are represented by the dynamic Java back-end is described in the next paragraph. The encoding of product `P2` is analogous to `P1`.

ABS Source Code

```
product P1 (F) {
   P2 delta D1 stateupdate U1;
}
product P2 (F,G) {
   P1 delta D2;
}
```

Generated Dynamic Java Code

```
1  package rdm;
2  public class P1_prod {
3     private static ABSDynamicProduct prod;
4     public static ABSDynamicProduct singleton() {
5        if (prod == null) {
6           ABSDynamicProduct prod = new ABSDynamicProduct();
7           prod.setName("P1");
8           prod.addReconfiguration(P2_prod.singleton(),
9              P1_P2_recf.singleton());
10       }
11       return prod;
12    }
13 }
14 public class P2_prod {...}
```

Figure 4.27: Dynamic representation of a DSPL product

Reconfigurations The code representation for ABS product reconfigurations is illustrated in Figure 4.28, based on the transition from product `P1` to product `P2`.

To represent the reconfiguration between `P1` and `P2`, a Java object of type `ABSDynamicReconfiguration` is created (line 6). It maintains references to the two products (lines 8–9), along with a list of deltas (line 10) and a state update (line 11); their runtime application performs the transition. The encoding of the reconfiguration `P2_P1` is analogous to `P1_P2`.

Generated Dynamic Java Code

```
1  package rdm;
2  public class P1_P2_recf {
3     private static ABSDynamicReconfiguration recf;
4     public static ABSDynamicReconfiguration singleton() {
5        if (recf == null) {
6           ABSDynamicProduct recf = new ABSDynamicReconfiguration();
7           recf.setName("P1->P2");
8           recf.setCurrentProduct(P1_prod.singleton());
9           recf.setTargetProduct(P2_prod.singleton());
10          recf.setDeltas(Arrays.asList(deltas.D1.D1_delta.singleton()));
11          recf.setUpdate(updates.U1_upd.singleton());
12       }
13       return recf;
14    }
15 }
16 public class P2_P1_recf {...}
```

Figure 4.28: Dynamic representation of a reconfiguration between two DSPL products

Deltas Figure 4.29 illustrates the representation of a delta in generated dynamic Java as an instance of the type `ABSDynamicDelta` (line 6). Here, the `D1` delta modifies class `C` by adding a new field and a getter method for the new field. Each class modifier is encoded as a class with a static `apply` method (line 17). The `apply` method applies the modifications to a copy of the targeted class `C` (`clsNext`). When the class duplicate is created, the original class's `nextVersion` field is set to point to the copy (line 20). By creating a linked list of all versions of a class, the class will always be able to find the newest version of itself. The modifications prescribed by the delta are then applied (lines 21–22). Finally, the old class object is replaced by the new class object (line 23). From this point on, new instances of `C` will be created using the updated version of the class.

ABS Source Code

```
delta D1;
modifies class Core.C {
   adds String s;
   adds String getS() {...}
}
```

Generated Dynamic Java Code

```
1  package deltas.D1;
2  public class D1_delta {
3     private static ABSDynamicDelta delta;
4     public static ABSDynamicDelta singleton() {
5        if (delta == null) {
6           delta = new ABSDynamicDelta();
7           delta.setName("D1");
8        }
9        return delta;
10    }
11    public void apply() {
12       deltas.D1.Core.C.Mod_b7f03583.apply();
13    }
14 }
15
16 package deltas.D1.Core.C;
17 public class Mod_b7f03583 {
18    public static void apply() {
19       ABSDynamicClass cls = Core.C_c.singleton();
20       ABSDynamicClass clsNext = cls.createNextVersion();
21       clsNext.addField("s", new ABSField() { /* Override init */ });
22       clsNext.addMethod("getS", new ABSClosure() { /* Override exec */ });
23       Core.C_c.setInstance(clsNew);
24    }
25 }
```

Figure 4.29: Dynamic Java encoding of a delta

State Updates Figure 4.30 shows the generated code used for updating each instance of class `C`. This object update is executed as a task of the COG of each object of class `C`. To ensure that updates are applied in the sequence that they were deployed, updates and objects bear version numbers. The object's version is matched against the update's version as part of the update guard (line 4). Then the object's class pointer is updated to point to a newer instance of `ABSDynamicClass` (lines 5–6) and the object's version number is incremented

ABS Source Code

```
stateupdate U;
objectupdate C {
   await(guard);
   classupdate;
   s = intToString(x);
}
```

Generated Dynamic Java Code

```
1  void applyObjectUpdate() {
2     ABSDynamicClass cls = this.getClazz();
3     Int updateVersion = 1;
4     ABSRuntime.await(guard && this.getVersion() == updateVersion);
5     ABSDynamicClass nextcls = cls.getNextVersion();
6     this.setClazz(nextcls);
7     this.incVersion();
8     this.addField("s");
9     this.setFieldValue("s", (ABSString)ABS.StdLib.intToString_f.apply(x));
10 }
```

Figure 4.30: Dynamic Java encoding of an object update

(line 7). Finally a new field `s` is added and assigned a `String` value based on the `intToString` function (which is part of the ABS standard library).

4.6.4 Usage

To generate Java bytecode using the dynamic ABS Java back-end, the ABS compiler is invoked using the `-dynamic` switch. For example, the following command generates Java code for the Chat SPL (in the `gen` directory):

```
java -cp absfrontend.jar abs.backend.java.JavaBackend -d gen \
    -dynamic ChatPL.abs
```

To execute the code generated from the Chat SPL example, one can use the following command line; the `-dynamic` switch enables the dynamic ABS runtime:

```
java -cp gen:absfrontend.jar ChatPL.Main -dynamic
```

These tasks can be performed equally using the Eclipse IDE with the ABS plugin installed.

4.7 Related Work

Our work on extending the ABS language framework to support DSPL development relates to research in the fields of dynamic software product lines and dynamic software updating.

4.7.1 Dynamic Software Product Lines

While traditional SPL engineering approaches are limited to static configuration of products, DSPL extend configurability to the runtime. A large portion of DSPL research is concerned with the delineation of the field itself (principles, properties, challenges) [106, 57, 112, 66]. Our understanding of DSPL concepts is largely based on this research, but we focus on providing a language and tool implementation that support the development of DSPL.

Context-oriented programming [36] and dynamic aspect-oriented programming [42] models have been used in connection with feature models as variability mechanisms for DSPL.

Feature- and delta-oriented programming (FOP and DOP) have both been instrumented to support runtime (re)binding of features. Rosenmüller et al. [97, 98] use FOP to statically compose sets of features called *dynamic binding units*, which can be switched on and off during runtime. Technically this is achieved by using the decorator design pattern to add behaviour to objects. A binding unit adds feature-specific behaviour by decorating the relevant classes. In contrast, we follow the DOP approach, which uses deltas to define (more general) program transformations. Our deltas exist and can be applied at runtime. We include a language feature that allows the developer to explicitly define how to transform the state of the program upon dynamic feature reconfiguration.

DOP has been also applied recently to dynamic SPL [40, 39, 62] in the sequential setting of the DeltaJ language. In DeltaJ the reconfiguration space is formalised by an automaton and a `reconfigure` instruction specifies safe reconfiguration points in the program. In contrast, we describe the reconfiguration semantics in a concurrent setting based on active objects, where updates are incremental rather than global. Instead of quiescent program update locations, we use update-specific safety guards. (See Section 4.3.4 for an encoding of `reconfigure` into ABS.) Additionally we provide a concrete implementation.

The ABS component model [78] describes a system in which objects communicate via input and output ports. A `rebind` instruction specifies dynamic reconfiguration points in the program. Methods can be annotated with the

keyword `critical`, which ensures that, during the execution of that method, the output ports of the object will not be modified. The reconfiguration semantics of ABS component systems are thus similar to DeltaJ, as described above.

Some studies investigate DSPLs from an architectural perspective. These generally do not approach low-level issues such as safe update points or state transfer. Oreizy et al. [91] propose an architectural model that is deployed with the system and used as a basis for dynamic changes along with an imperative language for modifying architectures. Weyns et al. [121] provide an architectural viewpoint and a framework for dynamically updating the SPL architecture.

4.7.2 Dynamic Software Updating

The traditional approach to updating a software system involves taking the system offline, updating, and restating it. However, the delay associated with this approach is sometimes unacceptable. Research into Dynamic Software Updating (DSU), sometimes called live, online or hot updating, focuses on the safety and timing of dynamic updates.

In our system, the liability to ensure that the system is in a consistent state during and after a dynamic update is assigned to the user, who can control the timing of updates of individual objects using an **await** guard (cf. Section 4.3.3). In other words, the system does not by itself guarantee that the update is safe. A range of studies propose update criteria that guarantee safety. We survey these for future reference.

Kramer and Magee [75] lay down a formal basis for safe dynamic reconfiguration of distributed programs. They introduce the notion of quiescence; a process in a quiescent state is both passive and has no outstanding messages to accept and service. For a dynamic reconfiguration to leave the system in a consistent state, all involved processes need to be quiescent. Achieving quiescence is a rather disruptive task: the process to be updated has to be put in a passive state along with all processes that are directly or indirectly capable of exchanging messages with it.

Vandewoude et al. [116] introduce the notion of tranquility, a more relaxed criterion for updatability, which, however, might not be reachable in bounded time. We include these references here for future reference; our update mechanism currently does not implement any notion of globally quiescent states, therefore consistency cannot be guaranteed.

Gupta et al. [55] present a notion of validity of a dynamic change, defined as the guarantee that a process that is updated will eventually reach a reachable state

of the updated program. They develop sufficient conditions for ensuring validity for a simple sequential language. Their results are not directly applicable in the concurrent context of ABS.

Several DSU systems propose the automatic derivation of safety constraints for applying updates to single-threaded systems [65, 110, 111]. When it comes to concurrent systems, such constraints may be both too broad to ensure timely application of updates, and generally insufficient to ensure safety [55]. This problem can be addressed by specifying safe update points in the code of each thread [89, 59]. The Proteus calculus [110] performs static updatability analysis to label program points at which updates could be applied. Our approach is to specify safe update conditions attached to the update code itself; this is arguably more flexible because it allows the safety criteria to be tailored specifically to each update.

Timing incremental updates is also challenging from the perspective of preserving type safety when objects communicate with each other during the reconfiguration period. Johnsen et al. [70] use type analysis to synchronise the updates of dependent objects. Wernli et al. [119, 120] introduce first-class contexts that represent different versions of a system; these are kept mutually compatible with the help of bidirectional transformations. Our system currently lacks the ability to generally ensure type safety over the reconfiguration period.

A major focus of existing DSU solutions is the application of bug fixes or evolutionary improvements to running systems. POLUS [25], for example, is a system that supports dynamic updating of single and multi-threaded systems written in C. Updates are patches that are externally generated based on the code difference between versions. We focus on modelling dynamic SPLs, which are reconfigurable according to a variability model; our updates are therefore part of the code and supported by the language.

Another important challenge when building a DSU system is how to transfer the state when updating objects. Approaches range from simply preserving the values of unaltered fields and initialising new fields with default values [24, 111, 124] to fully automated approaches [50, 82] based on analysing the program code or heap. Our solution automates the value transfer of unaltered fields but allows the user to specify state transformation functions.

4.8 Summary

This chapter introduces an extension of the ABS language and tool suite that adds support for modelling, implementing and executing dynamic software

product lines. This framework includes language constructs for specifying the runtime variability model, a meta-language to control reconfiguration from within the running model, and the implementation of both a runtime environment that readily supports runtime reconfiguration and a compiler that generates reconfigurable Java code. Concurrent systems developed in ABS are reconfigured incrementally, without the need for global quiescent states.

Chapter 5

Conclusion

5.1 Summary of Contributions

This section summarises the results of our research and highlights the contributions to the field of software product lines and software diversity modelling in general.

5.1.1 Modelling SPL behaviour with Feature Nets

The first contribution of this thesis is a modelling formalism based on Petri nets. While Petri nets are used to express the behaviour of single systems, *feature nets* enable the specification of the behaviour of an entire software product line (a set of systems) in one single, concise model. Feature nets model the diversity among variants of the SPL using the notion of feature. The behaviour of a feature net is conditional on the features appearing in the product line.

The feature nets formalism addresses the problem that many variability-intensive systems are safety-critical and thus their behaviour needs to modelled with rigour and is subject to verification. Feature nets support *modular* modelling, meaning that a typically large SPL can be modelled incrementally, e.g. one feature at a time. By following certain modelling criteria, the behaviour of the small individual nets is guaranteed to be preserved when these are combined together to a model of the entire SPL.

5.1.2 Developing and Executing (D)SPLs with ABS

Our second contribution is the extension of the ABS language and tool suite with variability modelling capabilities, thus creating the ABS integrated development environment for software product lines. The ABS IDE supports the modelling of SPLs both in the problem and in the solution domain, and establishes a connection between these two. This connection ensures that the problem and solution domain variability models stay consistent throughout the development life-cycle, and enables the automation of the application engineering process: when the user selects a particular product based on a set of desired features, the appropriate software product is automatically generated by the ABS compiler.

Our final contribution belongs to the domain of dynamically adaptable systems. We extend the ABS language and tool suite to add support for modelling, implementing and executing dynamic software product lines. This framework includes language constructs for specifying the runtime variability model, a meta-language to control reconfiguration from within the running model, and the implementation of both a runtime environment that readily supports runtime reconfiguration and a compiler that generates reconfigurable Java code. Concurrent systems developed in ABS are reconfigured incrementally, without the need for global quiescent states.

5.2 Future Work Directions

To consolidate and advance the research summarised in this dissertation, several directions for future work should be considered.

5.2.1 Feature Nets

For feature nets, the most important direction for future work is exploring the possibilities of analysis and verification. Analysis techniques for Petri nets, that is, verification of behavioural properties such as reachability, boundedness, liveness and reversibility [84] are an adequate starting point but such properties need to be interpreted in terms of their applicability and relevance to the domain of products and product lines. The practical applicability of the modular modelling FN techniques that we propose also needs closer examination, especially with respect to scalability. This could be accomplished by a case study involving more substantial SPL models.

5.2.2 ABS Variability Modelling

While the variability modelling facility of ABS has been designed around requirements and case studies provided by the HATS industrial partners, further empirical evaluation will bring more insight with respect to how the delta-oriented development paradigm helps modelling variable systems. This is especially true with regard to establishing best practices and recommended patterns of good delta design.

The dynamic reconfiguration facility of ABS needs to address the question of update safety. As distributed objects cannot be updated all at once without effectively halting the system, it is important to ensure that, in the course of an update, objects in the old processing state and those already in the updated processing state can interact seamlessly. A possible solution could be based on bidirectional transformation of state [120]. ABS already gives the user the possibility to specify a bidirectional correspondence between the state before and after a reconfiguration by defining a state update for each transition in the reconfiguration decision model. However, many more details need to be solved in this context, such as when is it safe to remove old class versions and fields.

Future work also needs to focus on automating tasks that currently require the user's intervention, such as the inference of dynamic deltas. Extending the support for SPL model evolution by developing a language for meta-variability modelling is another prospect. Finally, the performance and scalability of our tools and that of dynamic SPLs developed using our tools needs to be examined and, without doubt, improved.

Appendix A

Proofs (Chapter 2)

A.1 Proof of Theorem 2.5.12

Proof. Assume that Equation (2.1) holds for $N \oplus D = N'$. We show that the core behaviour is preserved, i.e., that $\forall FS \subseteq F \cdot \text{Beh}(N \downarrow FS) =_T \text{Beh}(N' \downarrow FS)$. Observe that, for every $FS \subseteq F$, $FS \cap (F_d \backslash F) = \emptyset$. Hence, by assuming Equation (2.1) we conclude that for every arc $(x, y) \in R_I$, $FS \not\models f(x, y)$. Therefore the traces $t \in \text{Beh}(N \downarrow FS)$ coincide with the traces of $\text{Beh}(N' \downarrow FS)$ with respect to the transitions of N. □

A.2 Proof of Theorem 2.5.13

Proof. Let $FS \subseteq F \cup F_d$, and $\mathcal{B}$ the bisimulation described by Theorem 2.5.13. We write M^N, M^D, and M^I to denote the markings M restricted to the places of N, D, and the interface of D, respectively.

We prove safety, while liveness can be shown in an analogous way, because $\mathcal{B}$ is symmetric. We show by induction the following property. Assume that $tr = t_1 \cdots t_n$ is a trace both in $\text{Beh}(N \oplus D \downarrow FS) \upharpoonright (T_d \backslash T_I)$ and in $\text{Beh}(D \downarrow FS) \upharpoonright (T_d \backslash T_I)$, ending in marking M and L respectively, and $M^N \mathcal{B} L^I$. Then for every transition t such that $(tr \cdot t) \in \text{Beh}(N \oplus D \downarrow FS) \upharpoonright (T_d \backslash T_I)$ ending in marking M', it also holds that $(tr \cdot t) \in \text{Beh}(D \downarrow FS) \upharpoonright (T_d \backslash T_I)$ ending in marking L' and $M'^N \mathcal{B} L'^I$. We now distinguish three scenarios for t.

1. $t \in T_d \backslash T_I$. t can also be performed by D. The possible problem is when places in ${}^{(FS)}t$ or $t^{(FS)}$ are in the interface. However, the property of $\mathcal{B}$ described by Theorem 2.5.13 guarantees that the shared markings between N and the interface of D have the same tokens for the shared places. The final marking in this case will still be in the bisimulation, i.e., the final marking L' from D will preserve the number of tokens in the M'^N and L'^I. Hence, by Equation (2.4) we conclude $M'^N \ \mathcal{B} \ L'^I$, and using induction hypothesis we conclude also that $(tr \cdot t) \in \text{Beh}(D{\downarrow}FS) \upharpoonright (T_D \backslash T_I)$.

2. $t \in T \cap T_I$. t is a transition from N and from D. Then the bisimulation gives that $t \ \mathcal{B} \ t$ and $L^I \xrightarrow{t,FS} L'^I$ is a firing from D, where $M^N \ \mathcal{B} \ L'^I$. Using the induction hypothesis we conclude that also $(tr \cdot t) \in \text{Beh}(D{\downarrow} FS) \upharpoonright (T_D \backslash T_I)$.

3. $t \in T \backslash T_I - t$ is a transition from N but not from D. Since $\mathcal{B}$ is a weak bisimulation and $M^N \ \mathcal{B} \ L^I$, then the interface of D can perform zero or more transitions in T_I (hence not visible when restricting to $T_D \backslash T_I$) until a transition L'^I such that $M'^N \ \mathcal{B} \ L'^I$. Again, the induction hypothesis allows us to conclude that $(tr \cdot t) \in \text{Beh}(D{\downarrow}FS) \upharpoonright (T_D \backslash T_I)$.

By (1), (2), and (3), and because the empty trace is always globally safe, we conclude by induction that any trace in $\text{Beh}(N \oplus D{\downarrow}FS)$ is also in $\text{Beh}(D{\downarrow}FS)$, when restricted to $T_d \backslash T_I$.

□

Bibliography

[1] *The ABS Language Specification*, 2013. available at `http://tools.hats-project.eu/download/absrefmanual.pdf`.

[2] Agerwala, T., and Flynn, M. Comments on capabilities, limitations and "correctness" of Petri nets. In *1st annual symposium on Computer architecture Proceedings* (1973), ISCA '73, ACM Press, pp. 81–86.

[3] Agha, G. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[4] Albert, E., Arenas, P., Gómez-Zamalloa, M., and Wong, P. Y. aPET: A test case generation tool for concurrent objects. In *9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2013)* (2013), ACM Press.

[5] Apel, S., Janda, F., Trujillo, S., and Kästner, C. Model superimposition in software product lines. In *Theory and Practice of Model Transformations*, vol. 5563 of *LNCS*. Springer, 2009, pp. 4–19.

[6] Arif, T., de Boer, F., Helvensteijn, M., Villela, K., and Wong, P. Y. H. Evaluation of Modeling, Mar. 2012. Deliverable 5.3 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[7] Asirelli, P., Beek, M., Fantechi, A., and Gnesi, S. A logical framework to deal with variability. In *Integrated Formal Methods*, vol. 6396 of *LNCS*. Springer, 2010, pp. 43–58.

[8] Asirelli, P., Beek, M., Fantechi, A., and Gnesi, S. A compositional framework to derive product line behavioural descriptions. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, vol. 7609 of *LNCS*. Springer, 2012, pp. 146–161.

[9] Atkinson, C., Bayer, J., and Muthig, D. Component-based product line development: The KobrA approach. In *International Software Product Line Conference*, vol. 576 of *The Springer International Series in Engineering and Computer Science*. Springer, 2000, pp. 289–309.

[10] Baldan, P., Corradini, A., Ehrig, H., and Heckel, R. Compositional semantics for open Petri nets based on deterministic processes. *Mathematical Structures in Computer Science 15*, 01 (2005), 1–35.

[11] Batory, D., Sarvela, J. N., and Rauschmayer, A. Scaling stepwise refinement. *IEEE Transactions on Software Engineering 30* (2004), 355–371.

[12] Benavides, D., Batory, D. S., and Grünbacher, P., Eds. *International Workshop on Variability Modelling of Software-Intensive Systems* (Linz, Austria, Jan. 2010), vol. 37, Universität Duisburg-Essen.

[13] Benavides, D., Segura, S., and Ruiz-Cortés, A. Automated analysis of feature models 20 years later: a literature review. *Information Systems* (2010).

[14] Benavides, D., Trinidad, P., and Ruiz-Cortés, A. Automated reasoning on feature models. In *Advanced Information Systems Engineering*. Springer, 2005, pp. 491–503.

[15] Bencomo, N., Hallsteinsen, S., and Santana de Almeida, E. A view of the dynamic software product line landscape. *IEEE Computer 45*, 10 (Oct. 2012), 36–41.

[16] Berger, T., Rublack, R., Nair, D., Atlee, J. M., Becker, M., Czarnecki, K., and Wąsowski, A. A survey of variability modeling in industrial practice. In *International Workshop on Variability Modelling of Software-Intensive Systems* (2013), VaMoS '13, ACM Press, pp. 7:1–7:8.

[17] Berthelot, G. Transformations and decompositions of nets. *Petri Nets: Central Models and Their Properties* (1987), 359–376.

[18] Bettini, L., Damiani, F., and Schaefer, I. Implementing software product lines using traits. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (2010), SAC '10, ACM Press, pp. 2096–2102.

[19] Bettini, L., Damiani, F., and Schaefer, I. Compositional type checking of delta-oriented software product lines. *Acta Informatica 50*, 2 (2013), 77–122.

[20] Bjørk, J., de Boer, F., Johnsen, E., Schlatte, R., and Tapia Tarifa, S. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering* (2012), 1–15.

[21] Boucher, Q., Classen, A., Faber, P., and Heymans, P. Introducing TVL, a text-based feature modelling language. In Benavides et al. [12], pp. 159–162.

[22] Bracha, G., and Cook, W. Mixin-based inheritance. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (1990), OOPSLA/ECOOP '90, ACM Press, pp. 303–311.

[23] Bracha, G., and Ungar, D. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (2004), OOPSLA '04, ACM Press, pp. 331–344.

[24] Chen, H., Yu, J., Chen, R., Zang, B., and Yew, P.-C. Polus: A powerful live updating system. In *International Conference on Software Engineering* (2007), ICSE '07, IEEE Press, pp. 271–281.

[25] Chen, H., Yu, J., Hang, C., Zang, B., and Yew, P.-C. Dynamic software updating using a relaxed consistency model. *IEEE Transactions on Software Engineering 37*, 5 (Sept. 2011), 679–694.

[26] Clarke, D. *Quality Assurance for Diverse Systems.* Jan. 2011, ch. 5, pp. 27–37. Deliverable 1.2 of the EternalS Coordination Action (FP7-247758), supported by the 7th Framework Programme of the EC within the FET (Future and Emerging Technologies) scheme.

[27] Clarke, D., Diakov, N., Hähnle, R., Johnsen, E., Schaefer, I., Schäfer, J., Schlatte, R., and Wong, P. Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In *Formal Methods for Eternal Networked Software Systems*, vol. 6659 of *LNCS*. Springer, 2011, pp. 417–457.

[28] Clarke, D., Helvensteijn, M., and Schaefer, I. Abstract delta modeling. In *International Conference on Generative Programming and Component Engineering* (2010), GPCE '10, ACM Press, pp. 13–22.

[29] Clarke, D., Muschevici, R., Proença, J., Schaefer, I., and Schlatte, R. Variability modelling in the ABS language. In *International Symposium on Formal Methods for Components and Objects* (2011), vol. 6957 of *LNCS*, Springer.

[30] Classen, A., Boucher, Q., and Heymans, P. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming 76*, 12 (Dec. 2011), 1130–1143.

[31] Classen, A., Heymans, P., and Schobbens, P. What's in a feature: A requirements engineering perspective. In *Fundamental Approaches to Software Engineering* (2008), pp. 16–30.

[32] Classen, A., Heymans, P., Schobbens, P.-Y., and Legay, A. Symbolic model checking of software product lines. In *International Conference on Software Engineering* (2011), ICSE '11, ACM Press, pp. 321–330.

[33] Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., and Raskin, J.-F. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *International Conference on Software Engineering* (2010), ICSE '10, IEEE Press, pp. 335–344.

[34] Cordy, M., Classen, A., Heymans, P., Legay, A., and Schobbens, P.-Y. Model checking adaptive software with featured transition systems. In *Assurances for Self-Adaptive Systems*, vol. 7740 of *LNCS*. Springer, 2013, pp. 1–29.

[35] Cordy, M., Classen, A., Perrouin, G., Schobbens, P., Heymans, P., and Legay, A. Simulation-based abstractions for software product-line model checking. In *International Conference on Software Engineering* (2012), ICSE '12, IEEE Press, pp. 672–682.

[36] Costanza, P., and D'Hondt, T. Feature descriptions for context-oriented programming. In *International Workshop on Dynamic Software Product Lines* (2008), Lero, University of Limerick, pp. 9–14.

[37] Ţiplea, F.-L. *On conditional grammars and conditional Petri nets*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1994, pp. 431–455.

[38] Czarnecki, K., and Antkiewicz, M. Mapping features to models: A template approach based on superimposed variants. In *Generative Programming and Component Engineering*, vol. 3676 of *LNCS*. Springer, 2005, pp. 422–437.

[39] Damiani, F., Padovani, L., and Schaefer, I. A formal foundation for dynamic delta-oriented software product lines. In *International Conference on Generative Programming and Component Engineering* (2012), GPCE '12, ACM Press, pp. 1–10.

[40] Damiani, F., and Schaefer, I. Dynamic delta-oriented programming. In *International Software Product Line Conference* (2011), SPLC '11, ACM Press, pp. 34:1–34:8.

[41] Desel, J., and Esparza, J. *Free choice Petri nets*. Cambridge University Press, New York, NY, USA, 1995.

[42] Dinkelaker, T., Mitschke, R., Fetzer, K., and Mezini, M. A dynamic software product line approach using aspect models at runtime. In *Workshop on Composition and Variability* (2010).

[43] Eisenecker, U. W., Apel, S., and Gnesi, S., Eds. *International Workshop on Variability Modelling of Software-Intensive Systems* (Leipzig, Germany, Jan. 2012), ACM Press.

[44] Fantechi, A., and Gnesi, S. Formal modeling for product families engineering. In *International Software Product Line Conference* (2008), SPLC '08, IEEE Press, pp. 193–202.

[45] Farooq, U., Lam, C. P., and Li, H. Transformation methodology for UML 2.0 activity diagram into colored Petri nets. In *Advances in Computer Science and Technology* (2007), ACTA Press, pp. 128–133.

[46] Fischbein, D., Uchitel, S., and Braberman, V. A foundation for behavioural conformance in software product line architectures. In *International Workshop on the Role of Software Architecture in Analysis and Testing* (2006), ACM Press, pp. 39–48.

[47] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. *Design Patterns*. Addison-Wesley, 1994.

[48] Ghabri, M.-K., and Ladet, P. Dynamic Petri nets and their applications. In *International Conference on Computer Integrated Manufacturing and Automation Technology* (1994), pp. 93–98.

[49] Girault, C., and Valk, R. *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*. Springer, Secaucus, NJ, USA, 2001.

[50] Giuffrida, C., Kuijsten, A., and Tanenbaum, A. S. Safe and automatic live update for operating systems. In *International conference on Architectural support for programming languages and operating systems* (2013), ASPLOS '13, ACM Press, pp. 279–292.

[51] Giuffrida, C., and Tanenbaum, A. Safe and automated state transfer for secure and reliable live update. In *Workshop on Hot Topics in Software Upgrades* (2012), HotSWUp '12, pp. 16–20.

[52] Gomaa, H. *Designing Software Product Lines with UML.* Addison Wesley, 2004.

[53] Gruler, A., Leucker, M., and Scheidemann, K. Calculating and modeling common parts of software product lines. In *International Software Product Line Conference* (2008), SPLC '08, IEEE Press, pp. 203–212.

[54] Gruler, A., Leucker, M., and Scheidemann, K. Modeling and model checking software product lines. In *International Conference on Formal Methods for Open Object-based Distributed Systems* (2008), vol. 5051 of *LNCS*, Springer, pp. 113–131.

[55] Gupta, D., Jalote, P., and Barua, G. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering 22*, 2 (1996), 120–131.

[56] Hähnle, R. The abstract behavioral specification language: A tutorial introduction. In *Formal Methods for Components and Objects*, vol. 7866 of *LNCS*. Springer, 2013, pp. 1–37.

[57] Hallsteinsen, S., Hinchey, M., Park, S., and Schmid, K. Dynamic software product lines. *IEEE Computer 41*, 4 (Apr. 2008), 93–95.

[58] Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G., and Svendsen, A. Adding Standardized Variability to Domain Specific Languages. In *International Software Product Line Conference* (2008), SPLC '08, IEEE Press, pp. 139–148.

[59] Hayden, C. M., Saur, K., Hicks, M., and Foster, J. S. A study of dynamic software update quiescence for multithreaded programs. In *Workshop on Hot Topics in Software Upgrades* (2012), HotSWUp '12, pp. 6–10.

[60] Heidenreich, F., and Wende, C. Bridging the Gap Between Features and Models. In *Workshop on Aspect-Oriented Product Line Engineering* (2007), AOPLE'07.

[61] Helvensteijn, M. Delta modeling workflow. In Eisenecker et al. [43], pp. 129–137.

[62] Helvensteijn, M. Dynamic delta modeling. In *International Workshop on Dynamic Software Product Lines* (2012), SPLC '12, ACM Press, pp. 127–134.

[63] Helvensteijn, M., Muschevici, R., and Wong, P. Y. H. Delta modeling in practice: a Fredhopper case study. In Eisenecker et al. [43], pp. 139–148.

[64] Hendrickson, S. A., and van der Hoek, A. Modeling product line architectures through change sets and relationships. In *International Conference on Software Engineering* (2007), ICSE '07, IEEE Press, pp. 189–198.

[65] Hicks, M., and Nettles, S. Dynamic software updating. *ACM Transactions on Programming Languages and Systems 27*, 6 (Nov. 2005), 1049–1096.

[66] Hinchey, M., Park, S., and Schmid, K. Building dynamic software product lines. *IEEE Computer 45*, 10 (Oct. 2012), 22–26.

[67] Hirschfeld, R., Costanza, P., and Nierstrasz, O. Context-oriented Programming. *Journal of Object Technology 7*, 3 (Mar. 2008), 125–151.

[68] Jayaraman, P., Whittle, J., Elkhodary, A., and Gomaa, H. Model composition in product lines and feature interaction detection using critical pair analysis. In *Model Driven Engineering Languages and Systems*, vol. 4735 of *LNCS*. Springer, 2007, pp. 151–165.

[69] Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., and Steffen, M. ABS: A core language for abstract behavioral specification. In *International Symposium on Formal Methods for Components and Objects* (2011), vol. 6957 of *LNCS*, Springer, pp. 142–164.

[70] Johnsen, E. B., Kyas, M., and Yu, I. C. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In *Formal Methods*, vol. 5850 of *LNCS*. Springer, 2009, pp. 596–611.

[71] Johnsen, E. B., Owe, O., Schlatte, R., and Tapia Tarifa, S. L. Dynamic resource reallocation between deployment components. In *International Conference on Formal Engineering Methods* (2010), vol. 6447 of *LNCS*, Springer, pp. 646–661.

[72] Kang, K. C., Cohen, S., Hess, J., Nowak, W., and Peterson, S. Feature-Oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-021, Carnegie Mellon University Software Engineering Institute, 1990.

[73] Kästner, C., Apel, S., and Kuhlemann, M. Granularity in software product lines. In *ICSE '08: Proceedings of the 30th international conference on Software engineering* (2008), ACM Press, pp. 311–320.

[74] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *European Conference on Object-Oriented Programming* (1997), vol. 1241 of *LNCS*, Springer, pp. 220–242.

[75] Kramer, J., and Magee, J. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering 16*, 11 (Nov. 1990), 1293–1306.

[76] Larsen, K., Nyman, U., and Wąsowski, A. Modal I/O automata for interface and product line theories. In *Programming Languages and Systems* (2007), vol. 4421 of *LNCS*, Springer, pp. 64–79.

[77] Larsen, K., and Thomsen, B. A modal process logic. In *Third Annual Symposium on Logic in Computer Science* (1988), IEEE Press, pp. 203–210.

[78] Lienhardt, M., Bravetti, M., and Sangiorgi, D. An object group-based component model. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation* (2012), vol. 7609 of *LNCS*, Springer, pp. 64–78.

[79] Lienhardt, M., and Clarke, D. Row types for delta-oriented programming. In Eisenecker et al. [43], pp. 121–128.

[80] Llorens, M., and Oliver, J. Structural and dynamic changes in concurrent systems: reconfigurable Petri nets. *IEEE Transactions on Computers 53*, 9 (Sept. 2004), 1147–1158.

[81] Lopez-Herrejon, R., Batory, D., and Cook, W. Evaluating support for features in advanced modularization technologies. In *European Conference on Object-Oriented Programming*, vol. 3586 of *LNCS*. Springer, 2005, pp. 169–194.

[82] Magill, S., Hicks, M., Subramanian, S., and McKinley, K. S. Automating object transformations for dynamic software updating. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (2012), OOPSLA '12, ACM Press, pp. 265–280.

[83] Moskowitz, H. R., Jacobs, B., and Firtle, N. Discrimination testing and product decisions. *Journal of Marketing Research (JMR) 17*, 1 (1980), 84–90.

[84] Murata, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE 77*, 4 (Apr. 1989), 541–580.

[85] Muschevici, R., Clarke, D., and Proença, J. Feature Petri Nets. In *Workshop on Formal Methods and Analysis in Software Product Line Engineering* (2010), vol. 2 of *SPLC '10*, Lancaster University, pp. 99–106.

[86] Muschevici, R., Clarke, D., and Proença, J. Executable modelling of dynamic software product lines in the ABS language. In *International Workshop on Feature-Oriented Software Development* (2013), FOSD '13, ACM Press. (to appear).

[87] Muschevici, R., Proença, J., and Clarke, D. Modular modelling of software product lines with Feature Nets. In *Software Engineering and Formal Methods* (2011), vol. 7041 of *LNCS*, Springer, pp. 318–333.

[88] Muschevici, R., Proença, J., and Clarke, D. Modular modelling of software product lines with feature nets. Tech. Rep. CW 609, KU Leuven, Belgium, 2011.

[89] Neamtiu, I., and Hicks, M. Safe and timely updates to multi-threaded programs. In *Programming Languages Design and Implementation* (2009), PLDI '09, ACM Press, pp. 13–24.

[90] Noda, N., and Kishi, T. Aspect-oriented modeling for variability management. In *International Software Product Line Conference* (2008), SPLC '08, IEEE Press, pp. 213–222.

[91] Oreizy, P., Medvidovic, N., and Taylor, R. N. Architecture-based runtime software evolution. In *International Conference on Software Engineering* (1998), ICSE '98, IEEE Press, pp. 177–186.

[92] Parnas, D. On the design and development of program families. *IEEE Transactions on Software Engineering 2* (1976), 1–9.

[93] Perrouin, G., Klein, J., Guelfi, N., and Jézéquel, J.-M. Reconciling automation and flexibility in product derivation. In *International Software Product Line Conference* (2008), SPLC '08, IEEE Press, pp. 339–348.

[94] Pohl, K., Böckle, G., and van der Linden, F. *Software Product Line Engineering*. Springer, 2005.

[95] Prehofer, C. Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience 13*, 6 (2001), 465–501.

[96] Rabino, S., and Moskowitz, H. Optimizing the product developing process using psychophysical scaling. *Journal of Business Research 10*, 3 (1982), 295–308.

[97] Rosenmüller, M. *Towards Flexible Feature Composition: Static and Dynamic Binding in Software Product Lines.* PhD thesis, Otto von Guericke University, Magdeburg, 2011.

[98] Rosenmüller, M., Siegmund, N., Pukall, M., and Apel, S. Tailoring dynamic software product lines. In *International Conference on Generative Programming and Component Engineering* (2011), GPCE '11, ACM Press, pp. 3–12.

[99] Schaefer, I. Variability modelling for model-driven development of software product lines. In Benavides et al. [12], pp. 85–92.

[100] Schaefer, I., Bettini, L., and Damiani, F. Compositional type-checking for delta-oriented programming. In *Aspect-Oriented Software Development Conference* (2011), AOSD '11, ACM Press, pp. 43–56.

[101] Schaefer, I., Bettini, L., Damiani, F., and Tanzarella, N. Delta-oriented programming of software product lines. In *International Software Product Line Conference* (2010), SPLC '10, Springer, pp. 77–91.

[102] Schaefer, I., and Damiani, F. Pure delta-oriented programming. In *International Workshop on Feature-Oriented Software Development* (2010), FOSD '10, ACM Press, pp. 49–56.

[103] Schaefer, I., Rabiser, R., Clarke, D., Bettini, L., Benavides, D., Botterweck, G., Pathak, A., Trujillo, S., and Villela, K. Software diversity: state of the art and perspectives. *Journal on Software Tools for Technology Transfer 14*, 5 (2012), 477–495.

[104] Schaefer, I., Worret, A., and Arnd, P.-H. A model-based framework for automated product derivation. In *International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE)* (2009).

[105] Schäfer, J., and Poetzsch-Heffter, A. JCoBox: Generalizing active objects to concurrent components. In *European Conference on Object-Oriented Programming* (2010), vol. 6183 of *LNCS*, Springer, pp. 275–299.

[106] Schmid, K., and Eichelberger, H. From static to dynamic software product lines. In *International Workshop on Dynamic Software Product Lines* (2008), Lero, University of Limerick, pp. 33–38.

[107] Schnoebelen, P., and Sidorova, N. Bisimulation and the reduction of Petri nets. In *Application and Theory of Petri Nets*, vol. 1825 of *LNCS*. Springer, 2000, pp. 409–423.

[108] Schwartz, B. *The Paradox of Choice: Why More is Less.* Harper Collins, 2003.

[109] Souissi, Y., and Memmi, G. Composition of nets via a communication medium. In *Advances in Petri Nets*, vol. 483 of *LNCS*. Springer, 1991, pp. 457–470.

[110] Stoyle, G., Hicks, M., Bierman, G., Sewell, P., and Neamtiu, I. Mutatis mutandis: Safe and flexible dynamic software updating. *ACM Transactions on Programming Languages and Systems 29*, 4 (Aug. 2007).

[111] Subramanian, S., Hicks, M., and McKinley, K. S. Dynamic Software Updates: A VM-centric Approach. In *Programming Languages Design and Implementation* (2009), ACM Press, pp. 1–12.

[112] Talib, M. A., Nguyen, T., Colman, A. W., and Han, J. Requirements for evolvable dynamic software product lines. In *International Workshop on Dynamic Software Product Lines* (2010), Lancaster University, pp. 43–46.

[113] Valk, R. Self-modifying nets, a natural extension of Petri nets. *Automata, Languages and Programming* (1978), 464–476.

[114] van der Linden, F., Schmid, K., and Rommes, E. *Software Product Lines in Action.* Springer, 2007.

[115] van Ommering, R. C. Software reuse in product populations. *IEEE Transactions on Software Engineering 31*, 7 (2005), 537–550.

[116] Vandewoude, Y., Ebraert, P., Berbers, Y., and D'Hondt, T. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering 33*, 12 (2007), 856–868.

[117] Voelter, M., and Groher, I. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *International Software Product Line Conference* (2007), SPLC '07, IEEE Press, pp. 233–242.

[118] von Rhein, A., Apel, S., Kästner, C., Thüm, T., and Schaefer, I. The PLA model: On the combination of product-line analyses. In *International Workshop on Variability Modelling of Software-Intensive Systems* (2013), VaMoS '13, ACM Press, pp. 14:1–14:8.

[119] Wernli, E., Gurtner, D., and Nierstrasz, O. Using first-class contexts to realize dynamic software updates. In *International Workshop on Smalltalk Technologies* (2011), IWST '11, ACM Press, pp. 2:1–2:11.

[120] Wernli, E., Lungu, M., and Nierstrasz, O. Incremental dynamic updates with first-class contexts. In *Technology of Object-Oriented Languages and Systems* (2012), vol. 7304 of *LNCS*, pp. 304–319.

[121] Weyns, D., Michalik, B., Helleboogh, A., and Boucke, N. An architectural approach to support online updates of software product lines. In *Working IEEE/IFIP Conference on Software Architecture* (2011), WICSA'11, pp. 204–213.

[122] Wong, P. Y., Albert, E., Muschevici, R., Proença, J., Schäfer, J., and Schlatte, R. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *Journal on Software Tools for Technology Transfer 14* (2012), 567–588.

[123] Wong, P. Y. H., Diakov, N., and Schaefer, I. Modelling adaptable distributed object oriented systems using the HATS approach: a Fredhopper case study. In *International Conference on Formal Verification of Object-Oriented Software* (2012), vol. 7421 of *LNCS*, Springer, pp. 49–66.

[124] Zero Turnaround. Live Rebel.

[125] Ziadi, T., Hélouët, L., and Jézéquel, J.-M. Towards a UML profile for software product lines. In *Software Product-Family Engineering*, vol. 3014 of *LNCS*. Springer, 2004, pp. 129–139.

Resume

Radu Muschevici was born in Cluj, Romania on 29 November 1973. He received a Bachelor of Science (B.Sc. Hons) degree from the University of Applied Sciences, Munich, Germany in 2005 and a Master of Science (M.Sc.) degree from Victoria University of Wellington, New Zealand in 2009. In 2009 he started doctoral studies in the iMinds-DistriNet research group at KU Leuven, Belgium under the supervision of Dave Clarke, focusing on programming language research in the context of the European Community project *Highly Adaptable and Trustworthy Software using Formal Models* (HATS). Aside from pursuing academic achievement, Radu has also gained extensive industry experience working for 8 years as a software engineer.

List of Publications

Journal Articles

- Peter Y.H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. In *Journal on Software Tools for Technology Transfer (STTT)*, 14:567–588, Springer, 2012.

International Conference Papers

- Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. Variability modelling in the ABS language. *International Symposium on Formal Methods for Components and Objects (FMCO)*, 3–5 October 2011, Torino, Italy. Volume 6957 of LNCS, pages 204–224, Springer, 2011.

- Radu Muschevici, José Proença, and Dave Clarke. Modular modelling of software product lines with feature nets. *Software Engineering and Formal Methods (SEFM)*, 14–18 November 2011, Montevideo, Uruguay. Volume 7041 of LNCS, pages 318–333, Springer, 2011.

- Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 19–23 October 2008, Nashville, TN, USA. Proceedings, pages 563–582, ACM Press, 2008.

- Christian Seifert, Barbara Endicott-Popovsky, Deborah Frincke, Peter Komisarczuk, Radu Muschevici, and Ian Welch. Identifying and analyzing

web server attacks. *Fourth Annual IFIP WG 11.9 International Conference on Digital Forensics*, 27–30 January 2008, Kyoto, Japan. Volume 285 of IFIP International Federation for Information Processing, pages 151–161, Springer, 2008.

International Workshop Papers

- Radu Muschevici, Dave Clarke, and José Proença. Executable modelling of dynamic software product lines in the ABS language. *International Workshop on Feature-Oriented Software Development (FOSD)*, 26 October 2013, Indianapolis, IN, USA. Proceedings, ACM Press, 2013.
- Michiel Helvensteijn, Radu Muschevici, and Peter Y. H. Wong. Delta modeling in practice: a Fredhopper case study. *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, 25–27 January 2012, Leipzig, Germany. Proceedings, pages 139–148, ACM Press, 2012.
- Radu Muschevici, Dave Clarke, and José Proença. Feature Petri Nets. *Workshop on Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*, 13–17 September 2010, Jeju Island, South Korea. Volume 2 of SPLC '10, pages 99–106, Lancaster University, 2010.

Technical Reports

- Radu Muschevici, José Proença, and Dave Clarke. Modular modelling of software product lines with feature nets. Technical Report CW 609, KU Leuven, Belgium, 2011, available at `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW609.abs.html`.

Project Deliverables

- Dave Clarke, Stijn de Gouw, Reiner Hähnle, Einar Broch Johnsen, Ilham W. Kurnia, Radu Muschevici, Bjarte M. Østvold, José Proença, Ina Schaefer, Jan Schäfer, Martin Steffen, and Arild B. Torjusen. Full ABS Modeling Framework, March 2011. Deliverable 1.2 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

- Frank de Boer, Dave Clarke, Michiel Helvensteijn, Radu Muschevici, José Proença, and Ina Schaefer. Final Report on Feature Selection and Integration, March 2011. Deliverable 2.2b of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.
- Taslim Arif, Ilham W. Kurnia, Radu Muschevici, Keiko Nakata, José Proença, Andri Saar, Tarmo Uustalu, Karina Villela, and Yannick Welsch. ABS System Derivation and Code Generation, September 2012. Deliverable 1.4 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.
- Dave Clarke, Einar Broch Johnsen, Radu Muschevici, José Proença, Michiel Helvensteijn, and Michaël Lienhardt. Hybrid Analysis for Evolvability, December 2012. Deliverable 3.3 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.
- Mads Dam, Andreas Lundblad, Radu Muschevici, Karl Palmskog, and Thiago Henrique Burgos de Oliveira. Autonomous Evolving Systems, March 2013. Deliverable 3.5 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.
- Mads Dam, Einar Broch Johnsen, Michiel Helvensteijn, Radu Muschevici, Rudolf Schlatte, and Lizeth Tapia. Evolvability Final Report, March 2013. Deliverable 3.6 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

Other Publications and Manuscripts

- Reiner Hähnle, Mads Dam, Radu Muschevici, Ina Schaefer, and Jan Schäfer. HATS: Highly adaptable and trustworthy software using formal models. Research Project Symposium at the *European Conference on Object-Oriented Programming (ECOOP)*, 27 July 2011, Lancaster, UK.
- Reiner Hähnle, Richard Bubel, Dave Clarke, Mads Dam, Radu Muschevici, Ina Schaefer, and Jan Schäfer. Highly adaptable and trustworthy software using formal models. Peer-reviewed poster at the *European Conference on Object-Oriented Programming (ECOOP)*, 27 July 2011, Lancaster, UK.
- Radu Muschevici. Multiple Dispatch in Practice. M.Sc. Thesis, Victoria University of Wellington, New Zealand, 2009.
- Radu Muschevici. Implementierung einer Datenbankanwendung für die Verwaltung einer komplexen EDV-Infrastruktur im Studentenwerk München. Diploma Thesis, Fachhochschule München, Germany, 2005.

FACULTY OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE
SCIENTIFIC COMPUTING GROUP
Celestijnenlaan 200A box 2402
B-3001 Heverlee
radu.muschevici@cs.kuleuven.be
http://distrinet.cs.kuleuven.be